AFRL-IF-WP-TR-2004-1532

WEAPON SYSTEM SOFTWARE TECHNOLOGY SUPPORT (WSSTS)
Delivery Order 0008: Real-Time Java for Embedded Systems (RTJES)

Edward Pla

The Boeing Company
Phantom Works/Network Centric Operations
P.O. Box 516
St. Louis, MO 63166-0516

MARCH 2004

Final Report for 21 September 2000 – 30 March 2004

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

Appendices A and B have been submitted to IEEE for publication in the Proceedings of the 2003 Real-Time Technology and Applications Symposium. If published, IEEE may assert copyright. If so, the United States has for itself and others acting on its behalf an unlimited, nonexclusive, irrevocable, paid-up royalty-free worldwide license to use for its purposes.

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

/s/

MICHAEL T. MILLS
Project Engineer

/s/

JAMES S. WILLIAMSON, Chief
Embedded Information Systems Branch
Advanced Computing Division
Information Directorate

James S. Williamson
/s/

for EUGENE BLACKBURN, Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange and does not constitute approval or disapproval of its ideas or findings.

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
March 2004	Final	09/21/2000 – 03/30/2004

4. TITLE AND SUBTITLE
WEAPON SYSTEM SOFTWARE TECHNOLOGY SUPPORT (WSSTS)
Delivery Order 0008: Real-Time Java for Embedded Systems (RTJES)

5a. CONTRACT NUMBER
F33615-97-D-1155-0008

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
78611F

6. AUTHOR(S)
Edward Pla

5d. PROJECT NUMBER
3090

5e. TASK NUMBER
02

5f. WORK UNIT NUMBER
43

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Boeing Company
Phantom Works/Network Centric Operations
P.O. Box 516
St. Louis, MO 63166-0516

8. PERFORMING ORGANIZATION REPORT NUMBER
BOEING-STL-2004P0013

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7334

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2004-1532

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Real-Time Java for Embedded Systems (RTJES) Program identified features of Real-Time (RT) Java suitable for meeting the challenges of future embedded information systems and defining the requirements of these features within the infosphere domain. After an initial survey of RT Java implementations, this program benchmarked the first commercially available RT Java implantation to provide an early assessment of the suitability of RT Java in the distributed real-time embedded system domain. Benchmarking efforts focused on assessing the performance and determinism of systems using RT Java. Two sets of tests were used. One examined characteristics of individual Real-Time Specification for Java (RTSJ) features. The other investigated performance within an environment representative of an avionics application. The RTJES program also performed a laboratory demonstration that showed the operational benefit of RT Java for network-centric applications. The demonstration highlighted RT Java code mobility, distribution, and portability.

15. SUBJECT TERMS
Real-Time Java, distributed real-time embedded systems, RTJES

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael T. Mills
Unclassified	Unclassified	Unclassified	SAR	68	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-6548 x3583

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

TABLE OF CONTENTS

Section Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

1 Scope

1.1 Identification

This final technical report (TR) describes the research results of the Real-Time Java for Embedded Systems (RTJES) Program, Delivery Order 0008 under the Weapon System Software Technology Support (WSSTS) Program, Contract Number F33615-97-D-1155, for the Air Force Research Laboratory (AFRL).

1.2 Introduction

The Boeing Company has experimented with Real-Time Java[1] (RT Java) as part of the AFRL RTJES program. This 3 year program focused on the use of Java in hard and soft real-time embedded information technology system applications. This research identified features of the Java programming model that may be especially helpful in meeting the challenges of future embedded system programs, and explored the suitability of these features within these domains. In particular, the program aimed to experiment with Real-Time (RT) Java Virtual Machines (JVMs) which implement the Real-Time Specification for Java (RTSJ) developed under the Sun Microsystems Java Community Process (JCP). The Boeing Bold Stroke Software Architecture has been leveraged to form an experimentation foundation and benchmark for comparison of Java-based implementation approaches. [2,3,4,5]

1.3 Programmatics

The objective of the RTJES Program is to develop, demonstrate, and mature RT Java-based embedded information systems applications. The RTJES effort involves identifying features of RT Java suitable for meeting the challenges of future embedded information systems and defining the requirements of these features within the infosphere domain.

The RTJES Program requirements as stated in the "Statement of Work, Real-Time Java for Embedded Systems (RTJES), Revision A" are listed in the remainder of this section in *italics.* Following each requirement, a statement in regular font is provided showing compliance.

---

[1] Java and related names are trademarks of Sun Microsystems, Inc.
[2] Winter, Don C., "Modular, Reusable Flight Software For Production Aircraft", *AIAA/IEEE Digital Avionics Systems Conference* Proceedings, October, 1996, p. 401-406.
[3] Sharp, David C., "Reducing Avionics Software Cost Through Component Based Product Line Development", *Software Technology Conference*, April 1998.
[4] Doerr, Bryan S., and Sharp, David C., "Freeing Product Line Architectures from Execution Dependencies", *Software Technology Conference*, May, 1999.
[5] Sharp, David C., "Avionics Product Line Software Architecture Flow Policies", *AIAA/IEEE Digital Avionics Systems Conference*, October 1999.

3.1 REAL-TIME JAVA REQUIREMENTS ANALYSIS

3.1.1 The Contractor shall assist in the definition of requirements and validation of selected use cases for RT Java as it relates to embedded mission processing systems. This effort shall investigate several aspects of RT Java's runtime system and evaluate its suitability for real-time embedded information technology systems in areas of interest to the Government.

Requirements definitions for RT Java based on Boeing's experience with embedded mission processing systems are documented in the "Test Descriptions and Results for the Real-Time Java for Embedded Systems Program" document, Version 3.0, dated 17 April 2003. The requirements related to the RTSJ are stated in section 3 "RTJES-Related Requirements." The requirements related to RT Java application level requirements are stated in Section 5 "Application Requirements". Highlights of these requirements are provided in Section 2.4 "RTJES-Related Requirements" and section 2.5 "Application Requirements" of this final report.

3.1.2 The Contractor shall examine strategies for using the technologies and ensuring appropriate testing is available.

RT Java provides for aspect programming, code mobility, and portability. The strategies for using these technologies have been tested and demonstrated on the Insertion of Embedded Infosphere Support Technologies (IEIST) Demonstration on February 24, 2004 in St. Louis, MO.

3.1.3 The Contractor shall participate bi-monthly in selected working groups associated with the specification of RT Java

Boeing actively participated in the Distributed Real-Time Expert Group Java Specification Request (JSR)-50. Boeing attended the Kickoff meeting at MITRE on 18-19 December 2000; Second meeting in Bedford, MA, on 1-2 February 2001; and Boeing hosted the third meeting in St. Louis, MO, on 19-20 March 2001. Boeing also worked extensively with TimeSys Corporation during initial checkout of the RTSJ Reference Implementation.

3.1.4 The Contractor shall document the results of this effort to include observations on military use of emerging standards, activities performed in the context of these meetings to convey military needs, and recommendations for further investigations.

As part of the of the Distributed Real-Time Specification for Java Expert Group meeting in St. Louis on 19-20 March, presentations were provided by system developers from the F/A-18 and F-15 programs, and from a Huntsville, AL Phantom Works representative. Two technical papers, included in Appendix A and Appendix B of this final report, were published on the merits of the RTSJ standard for large-scale embedded systems mainly those found in military avionics mission computing systems.

3.2 DEVELOPMENT OF REAL-TIME JAVA-RELATED TECHNOLOGIES FOR MILITARY APPLICATION

3.2.1 ARCHITECTURE

3.2.1.1 The Contractor shall propose experiments, develop prototypes, and analyze test results associated with determining the architectural suitability of RT Java for embedded mission systems.

Experiments and prototypes are documented in "Test Descriptions and Results for the Real-Time Java for Embedded Systems Program" document, Version 3.0, dated 17 April 2003. The RTSJ-related testing is documented Section 4 "RTSJ Related Testing" and the application level testing is documented in Section 6 "Real-Time Embedded Application Testing." The results of these experiments are measured against the established requirements for embedded mission systems.

3.2.1.2 The Contractor shall evaluate and analyze the use of RT Java to provide adaptive software capabilities to real-time systems.

One part of adaptive software capabilities is the ability to efficiently support several different software features. As part of the benchmark experiments and lab demonstration, technologies associated with the Washington University (WU) Framework for Aspect Composition for an EvenT Channel (FACET) RT Event Channel and University of California, Irving (UCI) Zen RT CORBA were tested. These products rely on Aspect Oriented Programming (AOP) technology designed to provide maximum flexibility of programming with a minimal size memory footprint. At the time of our testing, the FACET RT Event Channel did not provide sufficient throughput performance for it to be included in our test suite. The Zen RT CORBA product was used in the lab demonstration and benchmark results are in included in Section 5 "Real-Time Java Laboratory Demonstration" of this final report.

A second part of adaptive software capabilities is the ability to support different operating systems. At the time of this project, only one commercially available RTSJ implementation (TimeSys JTime) was available that supported only one real-time operating system (TimeSys RT Linux).

3.2.1.3 The Contractor shall evaluate the performance gains of selected algorithms, implemented in RT Java, as compared to current programming languages.

During application tests documented in Section 6 "Real-Time Embedded Application Testing" of the "Test Descriptions and Results for the Real-Time Java for Embedded Systems Program" document, side-by-side comparisons are performed with RT Java, standard Java, and C++. These results are also summarized in Section 4 "Application Tests and Results" of this final report.

3.2.1.4 The Contractor shall examine, to the extent feasible, verification and validation methodologies and tools, determining applicability, value, level of effort to use, as well as determining areas that are not addressed and develop a plan for ensuring that appropriate testing is available.

The development productivity is documented in Section 6.2 “Developmental Testing” of the “Test Descriptions and Results for the Real-Time Java for Embedded Systems Program” document. A side-by-side comparison was performed with Real-Time Java and C++.

3.2.2 DISTRIBUTED PROCESSING

3.2.2.1 The Contractor shall prototype and analyze, to the extent feasible, distributed processing scenarios using a RT Java implementation.

The RTJES program teamed with the IEIST program to perform a laboratory demonstration on February 24, 2004. This demonstration successfully executed a distributed processing scenario using RT Java.

3.2.2.2 The Contractor shall perform, to the extent feasible, whitebox and blackbox verification using a commercial RT Java implementation, of the network interface, concurrency, priority management, and memory management features, as they relate to distributed processing, present within the commercial implementation.

In addition to the IEIST demonstration, distributed benchmarks were performed on RT Java portion of the scenario and have been documented in Section 5.3 “Distributed Benchmarks” of this final report.

3.2.2.3 The Contractor shall implement distributed processing scenarios using Real-Time CORBA (RT CORBA).

UCI Zen RT CORBA was integrated with RT Java in preparation for the IEIST demonstration.

3.3 DEMONSTRATIONS

3.3.1 DEMONSTRATION OF DISTRIBUTED RT JAVA USING CORBA

3.3.1.1 The Contractor shall demonstrate distributed RT Java within an embedded mission system application using RT CORBA compliant object request brokers as per section 3.2.2, to the extent feasible with available products.

The IEIST demonstrated using UCI Zen RT CORBA technology was demonstrated and witnessed by AFRL representatives on February 24, 2004 at the Boeing facilities in St. Louis, MO.

3.3.1.2 The Contractor shall document the results of the demonstration, including a description of the technology, its benefits and restrictions, and its use in the demonstration. The Contractor shall collect metrics, as defined by the contractor, to show the benefit of the demonstrated technology. Additionally, the contractor shall describe its method of deployment such that system analysts and developers can employ the technology in ongoing efforts.

The results of the IEIST Demonstration are documented in Section 5 "Real-Time Java Laboratory Demonstration" of this final report. Metrics on throughput and deterministic performance were collected and provided in this report. The method of deployment is documented in this report so that system analysts and developers can employ this technology in ongoing efforts such as future IEIST Demonstrations.

3.4 TECHNICAL INTERCHANGE MEETING

The Contractor shall plan for and document the results of technical interchange meetings, which shall be held at Wright-Patterson AFB and the contractor's facility. The kickoff meeting shall be held no later than one (1) month after the delivery order award at Wright-Patterson AFB. A subsequent meeting shall be held midway through the technical effort at the contractor's facility. The contractor shall conduct a final technical interchange meeting.

The following Technical Interchange Meetings (TIMs) were held at Wright-Patterson AFB and Boeing, St. Louis.

1) Kickoff meeting was held in Wright-Patterson AFB on November 14, 2000.
2) TIM 1 was held in Boeing, St. Louis on March 23, 2001.
3) TIM 2 was held in Wright-Patterson AFB on October 29, 2001.
4) TIM 3 was held in Boeing, St. Louis on May 10, 2002.
5) TIM 4 was held in Boeing, St. Louis on December 11, 2002.
6) TIM 5 was combined with an IEIST Review and held in Wright-Patterson AFB on September 16, 2003.
7) Final review was also combined with an IEIST Review and held in St. Louis on February 24, 2004.

1.4 Document Overview

The remainder of this report begins by providing in section 2 "Technical Approach" the technical approach associated with the RTJES program emphasizing the experimentation systems and the derived requirements from our experience with avionics military embedded systems. In Section 3 "RTSJ Testing" and Section 4 "Application Testing," these requirements are used to formulate the RTSJ "low-level" testing and application-level testing for RT Java. In Section 5 "Real-Time Java Laboratory Demonstration," the previous tests form the basis for the final lab demonstration with the inclusion of distributed benchmarking. Final summaries of the test and laboratory results are provided in Section 6 "Summary." Reference information associated with this final report is listed in Section 7 "Reference Documents." Included in the appendix are the technical journals that were accepted for publication by IEEE as a result of this program.

2 Technical Approach

Using our Bold Stroke experience with avionics large-scale embedded system, we selected a set of infrastructure requirements applicable to our domain. From these requirements, we determined which requirements were supported by Real-Time Java. For the applicable associated requirements, a set of benchmarks and a lab demonstration was developed to determine the suitability to Real-Time Java. The applicability of the requirements and suitability of the test results were documented. This process is illustrated in Figure 1.

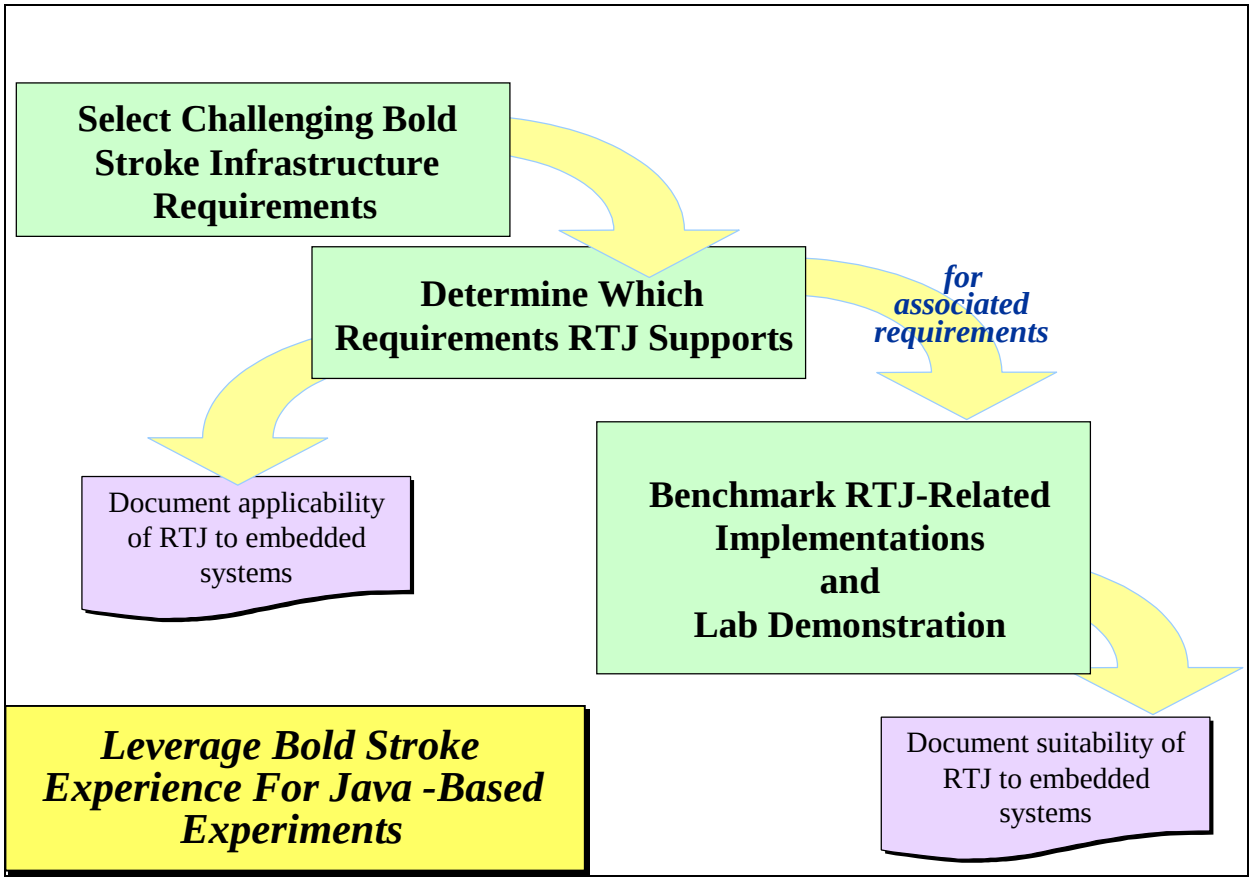


Figure 1. Technical Approach

Under the Java Community Process (JCP), each specification effort must provide three main items: (1) the specification itself, (2) a reference implementation of the specification, and (3) a test suite (referred to as the "technology compatibility kit (TCK)"). Since the TCK is required to test the functional semantics of compliant JVMs, the testing planned as part of RTJES focuses on assessment of nonfunctional properties of JVMs. The following key nonfunctional properties form the core experimentation areas:

1) *Performance:* investigating the throughput and latency associated with JVM capabilities.

2) *Determinism:* investigating the predictability associated with JVM capabilities, including jitter in various periodic or timed activities.

3) *Applicability to mission critical systems:* other areas of concern for mission-critical systems, including stability, functionality, and associated patterns of usage.

There are two categories of tests envisioned: (1) RTSJ-related tests, and (2) application-related tests. RTSJ-related tests investigate the qualities of specific RTSJ capabilities. Application-related tests investigate the qualities of integrated systems where RT JVMs are leveraged to perform representative mission critical software use cases. Application-related tests focus on assessing integrated behavior of multiple RTSJ capabilities as would be found in a final system. All of these requirements are documented in *Test Descriptions and Results for the Real-Time Java for Embedded System Program* report. This section will highlight those requirements that are most applicable to our current avionics environment.

2.1 Experimentation System

Early on in this program, the promises of at least a few RTSJ compatible JVMs were on the horizon. Figure 2 shows a list of potential RTSJ JVM alternatives that were considered for evaluation. Unfortunately, as time progressed only one commercially available RTSJ implementation was available on the market. The product is JTime from TimeSys.

Product	RTSJ Compliance	RTOS	HW Platform	BoldStroke Infrastructure Status on Platform	Comment on Java Interoperability with Non-Java Tasks	Comments
Timesys RT	Yes,	RTLinux	x86	Infrastructure ported to Redhat for SEC program. Would downloadable patch for conversion RTLinux. However, ACE/TAO not appear to have been ported RTLinux		Available Jan. 2001. look as good as thought because has not been ported RTLinux. Also the API is different than Linux and the two can't use of each other's resources.
OTI (IBM) Real-Java	Yes, although version does not implement entire	QNX Neutrino 2.0 Patch Level C (Can't be newer)	x86	Need to port to QNX ACE/TAO has been ported to Neutrino in		Downloadable beta version available. It only works an older version of Neutrino.
WindRiver Jworks	Not	VxWorks	PPC, etc.	Would be best comparison.	Other VxWorks tasks can run can be prioritized vs. the Java	WindRiver doesn't claim the JVM itself is real-Willing to work/share
NewMonics	It looks like it is compliant with J Consortium	VxWorks	PPC, etc.	Best Infrastructure	They support JNI and have their PNI, methods for Java threads communicated with non-Java It appears as though all Java run under one VxWorks task therefore, to the VxWorks they all appear to have the priority.	"Truly real-time Website says max 150 secs to preempt GC. Embedded Toolkit is a library that is patterned the specification developed by the Consortium. PERC runs a task under VxWorks RT Java tasks run within
Esmertek Jbed RTOS	Said yes but they actually on J-	Combined proprietary RTOS	PPC, etc.	No	Can call C++ functions but coexist with another	
EmWorks	Said yes eventually not presently sooner if	RTX	PPC, etc.	If they can port to VxWorks or (which we plan to port	RT Java threads are mapped underlying RTOS threads so, java threads can be prioritized vs. Java threads and vice-	Willing to make compliant and port VxWorks or QNX if

Table 1. Overview of Candidate JVMs

The next section describes the final configuration using JTime for both the hardware and software platform used for the testing and lab demonstration.

2.2 Hardware System

The same hardware, a Dell GX 150, was used for all benchmark development and execution. This computer has a single 1.2-GHz Pentium 4 processor. It has a 12-GB hard drive, 256 MB of RAM, and 256 KB of cache memory.

2.3 Software System

As of this writing, the only known planned commercial implementation of the RTSJ is JTime from TimeSys, for which we received a pre-release version. Prior to availability of this product, we developed tests and measured results on the openly available Reference Implementation (RI), also from TimeSys. The RI was designed to investigate and demonstrate the semantics of the RTSJ, not for acceptable run-time performance. Our prior RI benchmarking confirmed this.

The test platform was configured with Red Hat's Linux version 7.2, with real-time support provided by TimeSys Linux/NET for X86 UNI platform operating system extensions, version 3.1.214c, and TimeSys RT JVM version 3.5.3. Figure 3 lists the major software products used for the experiments, as well as the compiler options used that are associated with performance.

<table>
<tr><th></th><th>C++</th><th>RT Java</th><th>Java</th></tr>
<tr><td>Build Tools</td><td>GNU Make version 3.79.1</td><td colspan="2">Apache Ant Version 1.5.3</td></tr>
<tr><td>Compilers</td><td>GNU C++ version 3.1.1(<code>-O3 -g -fno -exceptions -fcheck-new</code>)</td><td colspan="2">Jikes version 1.14 (no compiler options used)</td></tr>
<tr><td rowspan="2">Run-Time Platform</td><td></td><td>TimeSys RT JVM version 3.5.3</td><td rowspan="2">Java J2SE with HotSpot version 1.4.1-2</td></tr>
<tr><td colspan="2">TimeSys Linux/NET for X86 UNI version 3.1.214c</td></tr>
</table>

Table 2. Software Packages

All Java tests were run with bytecodes generated by Jikes being interpreted in the JVM. For associated tests, the RT JVM was executed with a memory allocation pool of 50 MB (-Xms50M). The immortal memory size was set to 10 MB (IMMORTAL_SIZE=10000000).

To minimize nondeterministic system effects, all unnecessary operating services were stopped, virtual memory was disabled, and the system was rebooted in between tests.

The testing methodology for both the JTime from TimeSys and J9 from IBM is illustrated in Figure 4. The test application ran from a console window and output was recorded in

a test file. The J9 JVM did not improve significantly from its original implementation and was later dropped as a potential test platform.

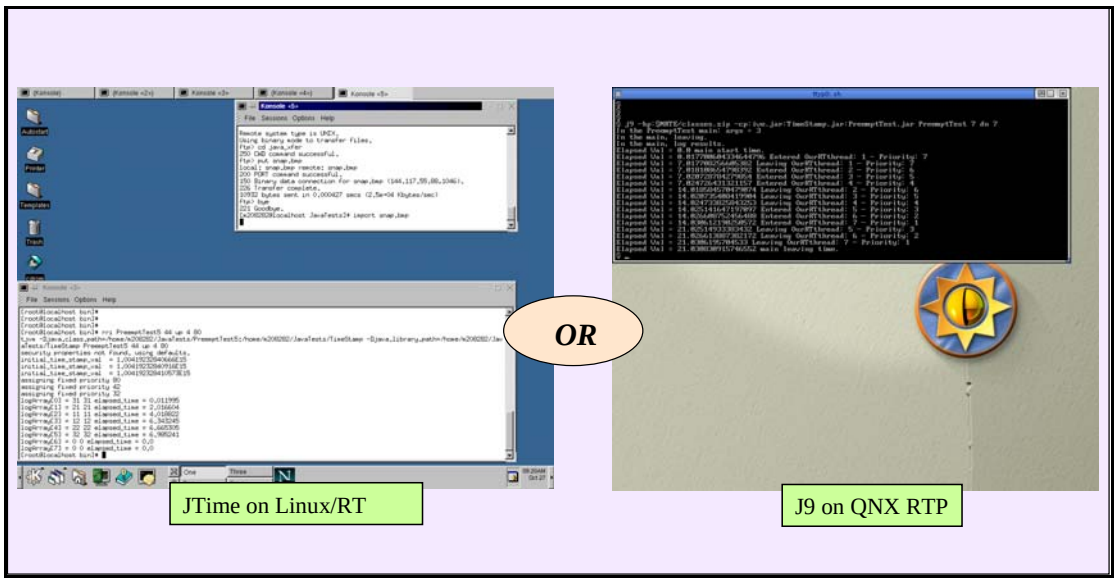


Figure 2. Test Methodology for JVMs

2.4 RTSJ-Related Requirements

The RTSJ capabilities supported by the RT JVMs exhibit a wide range of requirements. The requirements presented below come from requirements used on various Boldstroke and Boeing military projects. This section covers only the requirements for the most important RTSJ-related tests. The complete set of the RTSJ-related requirements can be found in the *Test Descriptions and Results for the Real-Time Java for Embedded System Program* report, Section 3 "RTJES-Related Requirements."

2.4.1 Thread

The thread requirements are concerned with consistency in the start and end time determinism of various thread and latencies in switching between threads.

Context Switch Latency - Context switch latency shall be less than 10 microseconds.

Thread throughput - Throughput in different threads shall not vary by more than 1 per cent for No Heap Real-Time Threads (NHRT) and Real-Time Threads (RT).

2.4.2 Scheduling

The scheduling requirements are concerned with the processor utilization times and latencies in executing the overrun handler.

Periodic Start of Frame Determinism – Jitter shall be within 1 per cent of the period. With a representative avionics-processing rate of 20 Hz, the maximum allowable jitter is 0.5 milliseconds.

Periodic End of Frame Determinism – Completion time differences shall be under 0.5 milliseconds. This represents 1 per cent of a 20-Hz frame.

Aperiodic End of Frame Determinism - Completion time differences shall be under 0.5 milliseconds. This represents 1 per cent of a 20-Hz frame.

2.4.3 Memory Management

The memory management requirements are concerned with the processor utilization times for entering and exiting Memory Areas, allocating objects, finalization performance, and garbage collection performance.

Instruction execution time in different Memory Areas - The throughput values shall be within 1 per cent across all memory areas.

Object allocation time in different Memory Areas - Average allocation time less than 2.0 microseconds/byte shall be acceptable.

Memory Area entry/exit times - Average memory area entry time shall be 100 microseconds or less. Average memory area exit time shall be 100 microseconds or less.

Jitter execution time performance - Jitter shall be less than 0.5 milliseconds for each memory area. This represents 1 per cent of a 20-Hz frame.

2.4.4 Synchronization

The synchronization requirements are concerned with measuring various software system latencies.

Priority inheritance latency - Priority latency shall be under 50 microseconds for boosting and unboosting priorities combined.

Synchronization latency - Synchronization latency shall be under 10 microseconds of overhead (difference between synchronized and non-synchronized).

Synchronization block latency - Synchronization latency shall be under 10 microseconds of overhead (difference between synchronized and non-synchronized block execution times).

2.5 Application Requirements

For this evaluation, we have captured key operational metrics based on our experience with the required run-time performance of a range of avionics systems. Since the RT JVM does not include specific support for distribution, these requirements are captured at the level of the complete software system and therefore apply primarily in the context of benchmark tests that are representative of full-scale single processor avionic mission computing systems.

2.5.1 Performance

Performance in this context refers to the capability of a software system to meet timing and sizing constraints imposed by the application functionality and the hardware and software operating environment.

Steady State Time - The software system shall support execution of multiple periodic rates of application components up to 20 Hz. The execution time required for infrastructure services (e.g. middleware, JVM, operating system) shall not exceed 25 per cent of the total processing time (when measured in a full-scale system that is fully exercising those services).

Memory Usage - The software system shall support execution within a memory limit of approximately 100 MB, including a full application load.

Duration of Operation - The software system shall support stable execution up to approximately 10 hours.

2.5.2 Determinism

Determinism in this context refers to the capability of a software system to have predictable resource utilization, regardless of speed.

Steady State Time - The variability in the start of periodic processing frames shall not exceed 1 per cent of the associated period. The variability in the time required to process the application within each execution rate shall not exceed 5 per cent of the associated period.

Memory Usage - The variability in the memory used by the software system during multiple identical runs shall not exceed 5 per cent.

Scalability - The software system will minimize the performance cost for adding software functionality to O (n log n).

3 RTSJ Testing

These tests focus on assessing the performance of specific RTSJ capabilities. These tests have been added to the RTJPerf open source RT JVM benchmarking suite established by Angelo Corsaro at the University of California, Irvine (UCI) [6] and Washington University in St. Louis. Taken together, tests were created to assess determinism, latency, and throughput associated with threads, scheduling, memory management, synchronization, time, timers, asynchrony, exceptions, and class loader and dynamic linking. Only the tests deemed of most importance are included here. A complete set of the tests can be found in the *Test Descriptions and Results for the Real-Time Java for Embedded System Program* report.

For each test, a description of the test is included, along with success criteria, and experimental results and analysis. The success criteria are based on requirements and experiences with avionic mission computing systems. While these criteria are intentionally domain specific, they do capture expectations for an important category of embedded systems. In all cases, the raw measured values are provided for comparison against criteria in other domains.

3.1 Throughput

The RTSJ introduces two new types of threads as shown in Figure 5: RT threads and No Heap RT (NHRT) threads. RT threads support, at a minimum, basic real-time preemptive scheduling. No Heap RT threads add the guarantee that execution will be independent of garbage collection but with the additional restriction that heap-based memory not be used. This section outlines tests assessing the throughput of these different thread types in varying execution environments.

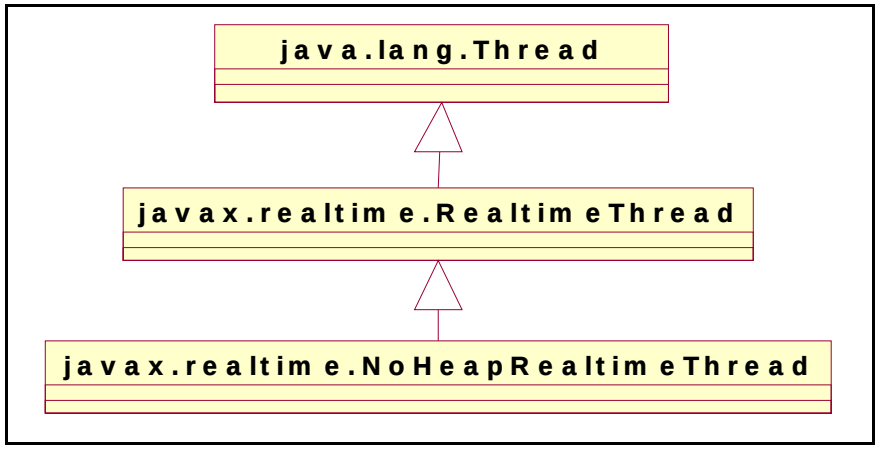


Figure 3. Thread Inheritance in RT Java

[6] A. Corsaro, D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems", Proceedings of the 8th IEEE Real-time Technology and Applications Symposium, September 2002.

3.1.1 Thread Throughput

Description: Record the execution time of a computationally intensive algorithm, representative of avionics mission computing processing, when run in different thread types: No Heap Real-Time Thread (NHRT), Real-Time Thread (RT), and java.lang.Thread. All thread types will be processing in a 20-Hz frame. In the thread's `run()` method, log timestamps before and after the algorithm executes. This computationally intensive algorithm is a flight controls algorithm that is CPU intensive and reads data from two different input files. The flight controls algorithm performs all of its memory allocation and reference storage upon initiation. The No Heap Real-Time Threads were executed using Immortal memory that has no garbage collection while the Real-Time Threads were executed from heap memory.

Success Criteria: Throughput in different threads shall not vary by more than 1 per cent. The test is illustrated in Figure 6.

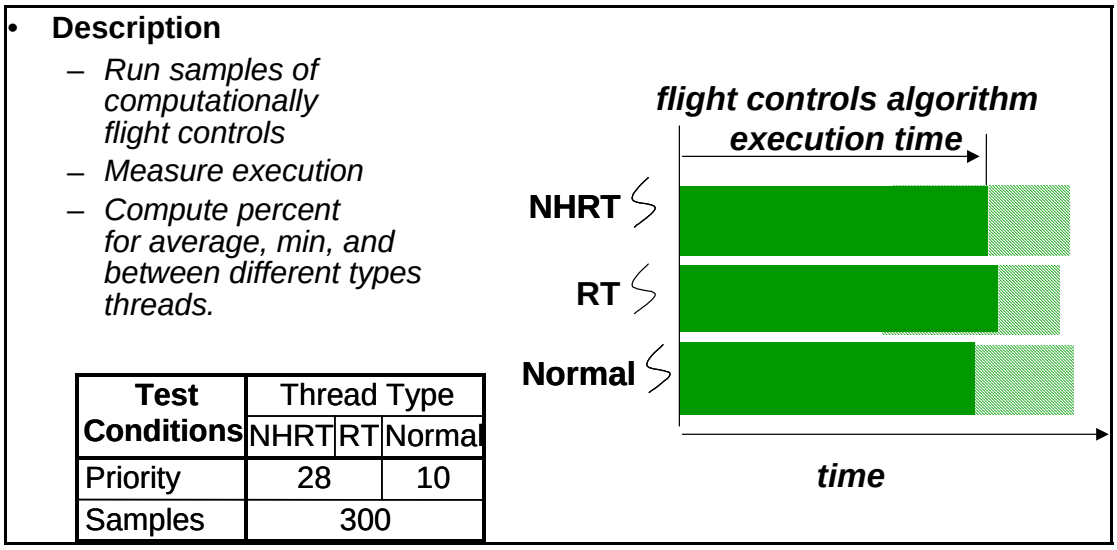
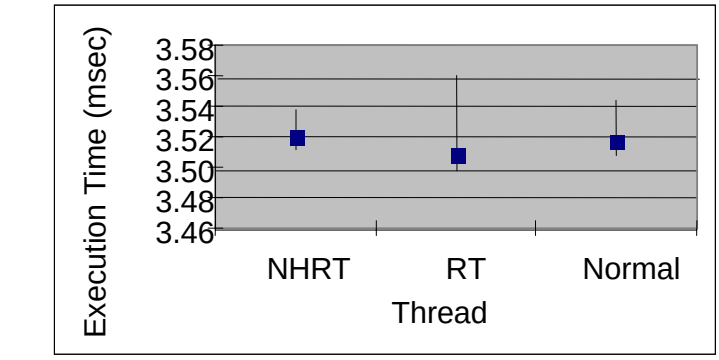


Figure 4. Execution with Different Thread Types

Results: The throughput for NHRT, RT, and normal java threads on the selected algorithm was comparable. The largest percentage time difference between the three thread types was 0.643 per cent for the NHRT and RT threads. See Figure 7 for more throughput comparisons between NHRT, RT, and normal threads.



	NHRT (ms)	RT (ms)	Normal (ms)	% Difference
Ave	3.5201	3.5087	3.5174	0.324%
Max	3.5372	3.5601	3.5439	0.643%
Min	3.5109	3.4977	3.5076	0.376%

Figure 5. Execution with Different Thread Types Results

3.1.2 Thread Throughput With Contending Background Threads

Description: Record the execution time of the same mission computing algorithm, when run with different scheduling parameters and competing threads. The algorithm is CPU intensive and executes in a NoHeapRealtimeThread with a priority of 260. Measure how the same functionality scheduled with varying numbers of lower priority threads behaves. The contending threads execute in RealtimeThreads with lower priorities. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20-Hz frame rate.

Success Criteria: Difference between the tests with no background threads and the tests with 15 background threads shall be less than 5 per cent. The test is illustrated in Figure 8.

- **Description**
  - *Run periodic samples of the computationally intensive flight controls algorithm with/without contending background thread.*
  - *Measure execution times*
  - *Compute percent difference between max and min execution time.*

Test	Test Case	
	No Contention	Contention
Duration	20	
NHRT	260	
Periodicity	100	
Number of RTs	0	15
Priority of RTs	N/A	250

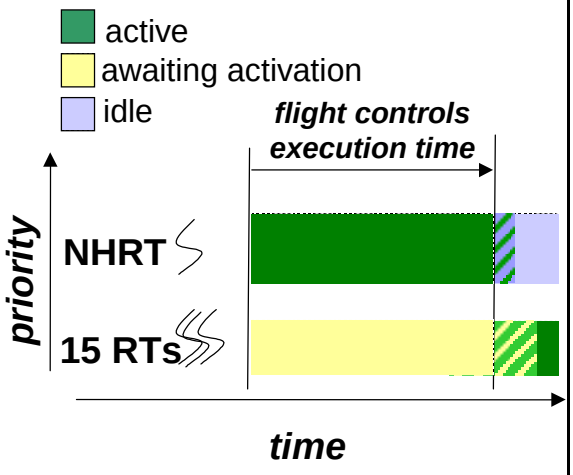
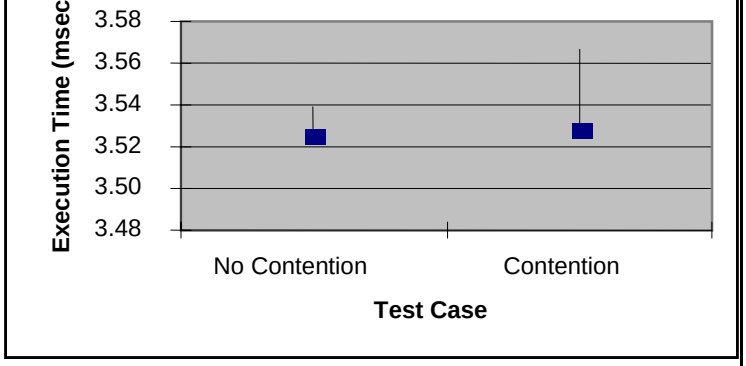


Figure 6. Execution with Contending Background Threads

Result: The first case schedules only the thread being analyzed, while the second case schedules the thread to be analyzed along with 15 background threads. The difference between the maximum, minimum, and average data points were all below 1 per cent. This meets our success criteria. See Figure 9 for detailed metrics.



	No Contention (ms)	Contention (ms)	% Difference
Avg	3.5253	3.5290	0.1039%
Max	3.5495	3.5669	0.7673%
Min	3.5218	3.5244	0.0756%
Jitter	0.0177	0.0424	

Figure 7. Execution with Contending Background Threads Results

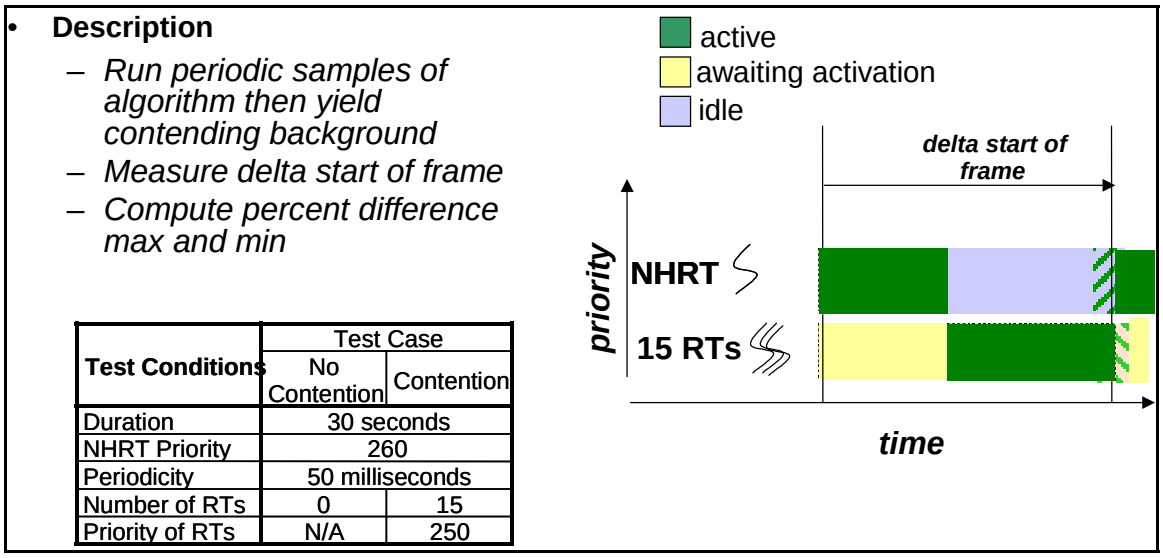
3.2 Determinism

The RTSJ provides direct support for initiating functionality that needs to be run at periodic intervals either via thread scheduling or via events driven by timers. This section outlines tests investigating timing jitter associated with initiating and completing periodic activities.

3.2.1 Periodic Start of Frame Determinism

Description: Using the `PeriodicParameters` class, establish a periodic thread. Immediately after the `waitForNextPeriod()` call, log a timestamp and calculate the time between invocations. Two tests were conducted. The first test was executed with only a single 20-Hz NoHeapRealtimeThread being analyzed, while the second test was executed with the 20-Hz NoHeapRealtimeThread thread being analyzed while another fifteen lower priority RealtimeThread threads were executing at a 20-Hz frame rate. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20-Hz frame rate.

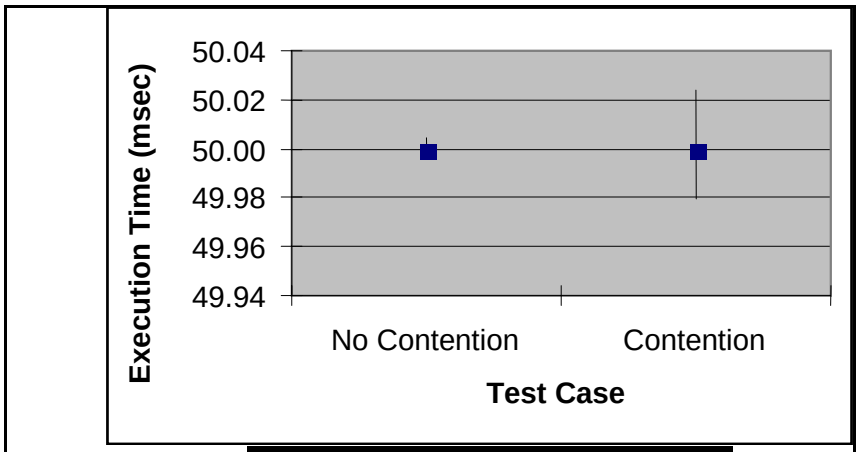
Success Criteria: Jitter shall be within 1 per cent of the period. With a representative avionics processing rate of 20 Hz, the maximum allowable jitter is 0.5 milliseconds. The test is illustrated in Figure 10.



Test Conditions	Test Case	
	No Contention	Contention
Duration	30 seconds	
NHRT Priority	260	
Periodicity	50 milliseconds	
Number of RTs	0	15
Priority of RTs	N/A	250

Figure 8. Periodic Start of Frame Determinism

Results: The maximum jitter in both tests easily surpassed the success criteria of 0.5 milliseconds. See Figure 11 for details.



	No Contention (ms)	Contention (ms)
Avg	50.0	50.0
Max	50.0048	50.0241
Min	49.9954	49.9794
Difference	0.0094	0.0447

Figure 9. Periodic Start of Frame Determinism Results

3.2.2 Periodic End of Frame Determinism

Description: Using the PeriodicParameters class, set up a periodic NoHeapRealtimeThread thread. Immediately after the `waitForNextPeriod()` call, execute an algorithm of significant duration but not longer than the period. After the algorithm completes, log a timestamp and calculate the difference between successive timestamps. Repeat with and without competing lower priority RealtimeThread threads as for the previous test. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20-Hz frame rate.

Success Criteria: Completion time differences shall be under 0.5 milliseconds. This represents 1 per cent of a 20-Hz frame.

Results: The maximum jitter for 0 and 15 competing threads was 0.0282 milliseconds and 0.1441 milliseconds, respectively, which easily met the success criteria. See Figure 12 for specific measurement results.

	0 Other Threads	15 Other Threads
Avg	50.0000	50.0000
Max	50.0153	50.0688
Min	49.9870	49.9247
Difference	0.0282	0.1441

Table 3. Frame Execution Times Measured at Frame Completion (milliseconds)

3.2.3 Periodic Event Determinism

Description: Measure the jitter in PeriodicTimer driven AsyncEvents. Immediately inside the handleAsyncEvent method, log a timestamp and calculate the time between invocations. The first test was executed with only a single 20-Hz NoHeapRealtimeThread being analyzed, while the second test was executed with the 20-Hz NoHeapRealtimeThread thread being analyzed while another fifteen lower priority RealtimeThread threads were executing at a 20-Hz processing rate also.

Success Criteria: Periodic event timing differences shall be under 0.5 milliseconds, 1 per cent of a 20-Hz (50 millisecond) frame. This test is illustrated in Figure 13.

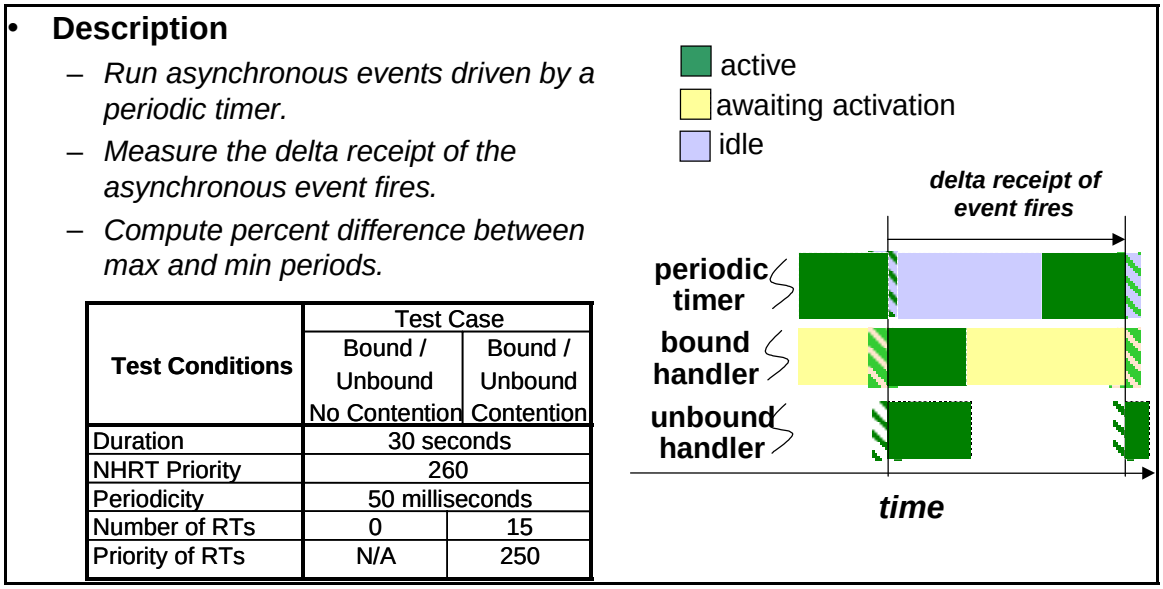


Figure 10. Periodic Event Determinism

Results: The first case was run with the AsyncEventHandler analyzing a single thread while the second case was executed with the AsyncEventHandler analyzing a single

thread with fifteen background threads. The third case was run with the BoundAsyncEventHandler analyzing one thread while the fourth case was executed with the BoundAsyncEventHandler analyzing a single thread with fifteen background threads. In all cases, the jitter met the success criteria. See Figure14 for more completion time comparisons with and without competing threads.

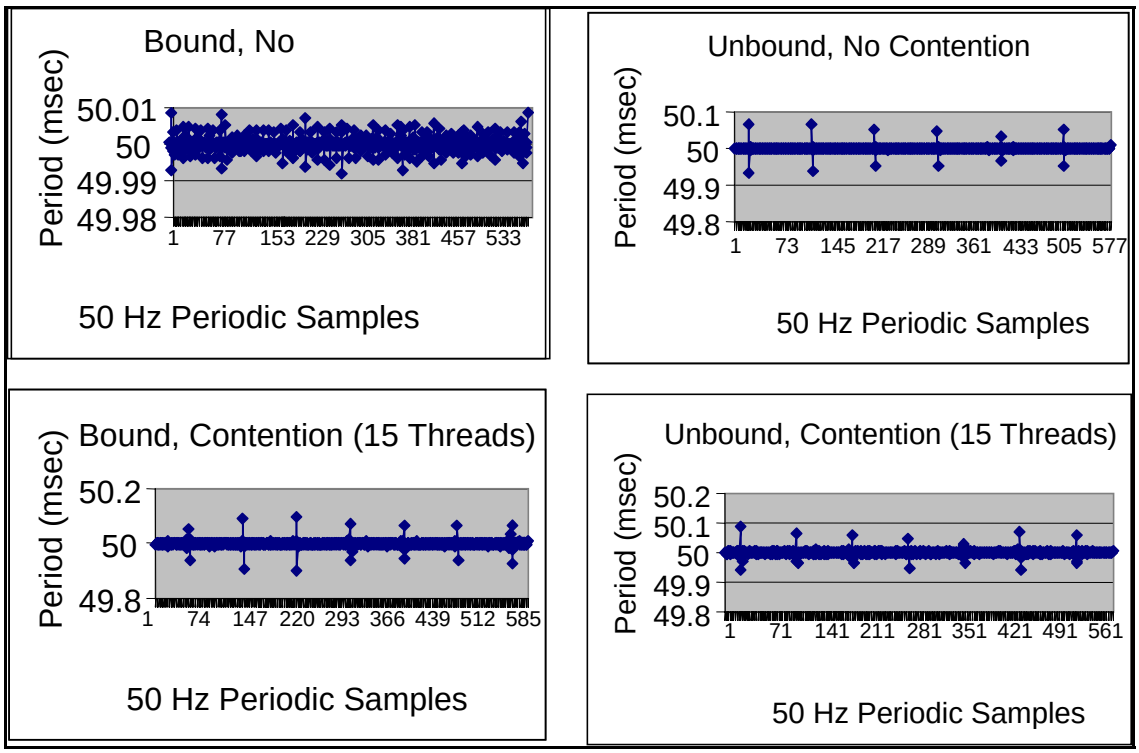


Figure 11. Periodic Event Determinism Results

3.3 Latency

This section details tests assessing delays associated with context switching, synchronization, and event delivery. The RTSJ supports event-based programming for two types of event handlers: bound and unbound. A bound event handler creates one thread that is permanently bound to the handler and remains active for all event fires. An unbound event handler creates a new thread with each event fire.

3.3.1 Context Switch Latency

Description: Initiate a high priority thread and a lower priority thread. Both threads will be executing at a 20-Hz frame rate. In the higher priority thread, log a timestamp before the `yield()` in the run method. In the lower priority thread, log a timestamp after the `yield()` in the run method. Then compute the latency between the higher priority thread's timestamp and the lower priority thread's timestamp.

Success Criteria: Context switch latency shall be less than 10 microseconds. This test is illustrated in Figure 15.

Results: The results show a median of approximately 2.1 microseconds. Some of the samples spike to 2.3-2.8 microseconds, probably indicating that some processing in addition to the context switch is being run following the `yield()` call. Even with this, however, the maximum time to switch between threads was roughly 2.8 microseconds, which is better than the 10 microsecond success criteria.

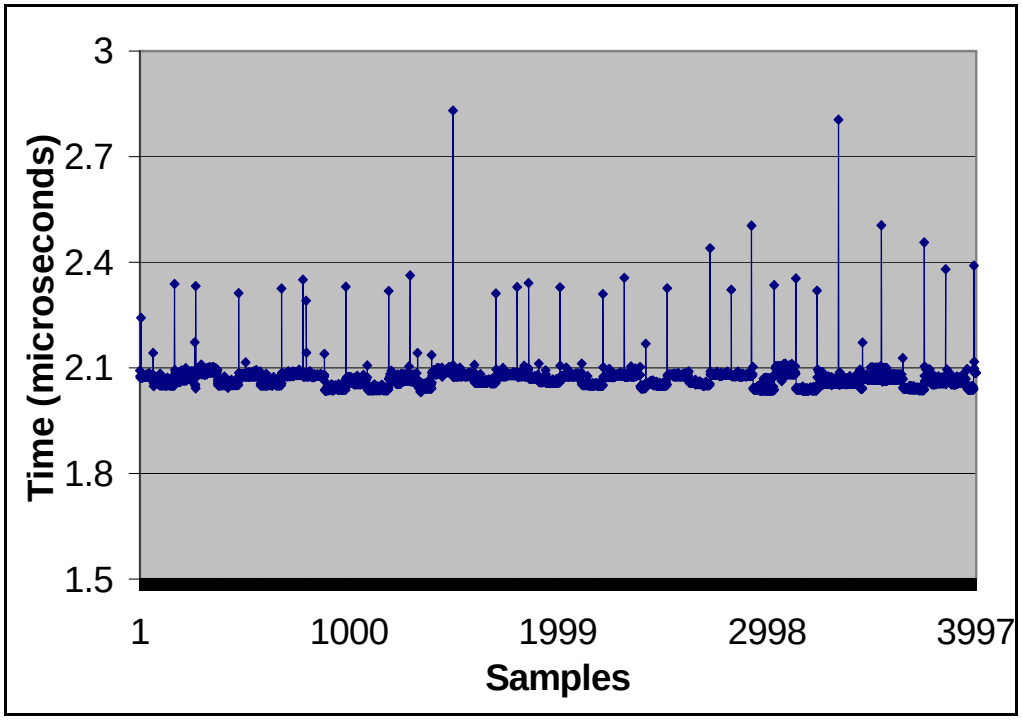


Figure 12. Context Switch Latency

3.3.2 Priority Inheritance Latency

Description: This test measures a relatively simple three thread case of priority inheritance. The low priority thread (LPT) starts and enters a synchronized method. While in that method, the medium priority thread (MPT) starts and preempts the LPT. While the MPT runs, a high priority thread (HPT) preempts the MPT and attempts to enter the same synchronized method the LPT presently has a lock on. According to priority inheritance, the LPT should get boosted up to the priority of the HPT so it can finish with the synchronized method, thus allowing the HPT to run as soon as possible. Log timestamps before and after the calls to the synchronized method. Also log timestamps at the first and last instructions inside the synchronized method. These timestamps are used to measure the boost, unboost, and total priority inheritance latency times. Each thread will execute at a 20-Hz frame rate.

Success Criteria: Priority latency shall be under 50 microseconds for boosting and unboosting priorities combined.

Results: For both cases the maximum latency was roughly 9.0 microseconds, thus the test passed the 50 microsecond success criteria. See Figure 16 for more priority inheritance boost, unboost, and total latencies.

	Boost	Unboost	Latency (Boost + Unboost)
Avg	5.1372	2.7735	7.9107
Max	6.2626	3.1574	8.9635
Min	4.4778	2.6517	7.1566
Delta	1.7849	0.5057	1.8070

Table 4. Priority Inheritance Latency (microseconds)

3.3.3 Synchronization Latency

Description: Record the time elapsed to enter a synchronized method versus a non-synchronized method. Log timestamps prior to the method call and once inside the synchronized and non-synchronized methods. Each thread will execute at a 20-Hz frame rate.

Success Criteria: Synchronization latency shall be under 5 microseconds of overhead (difference between synchronized and non-synchronized).

Results: The test was executed for the synchronized and normal method latency cases. For each case the latency differences were less than 2 microseconds, thus this test passes the 5 microsecond threshold. See Figure 17 for more synchronized and non-synchronized latencies.

	Non-Synchronized	Synchronized	Difference
Avg	1.3351	3.2385	1.9034
Max	1.7257	3.6962	1.9705
Min	1.3153	3.1975	1.8822

Table 5. Synchronization Latency (microseconds)

3.3.4 Event Latency

Description: Measure the latency from the firing of an AsyncEvent to the time it is handled. Log timestamps prior to the fire and once the event is handled. Each thread will execute at a 20-Hz frame rate.

Success Criteria: Event latency shall be under 100 microseconds. The test is illustrated in Figure 18.

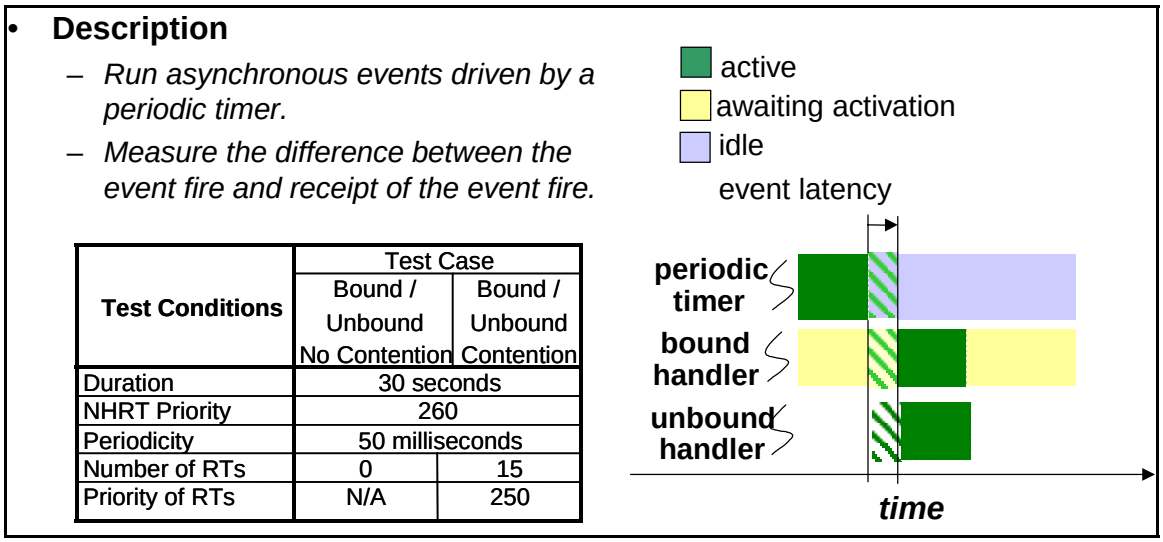
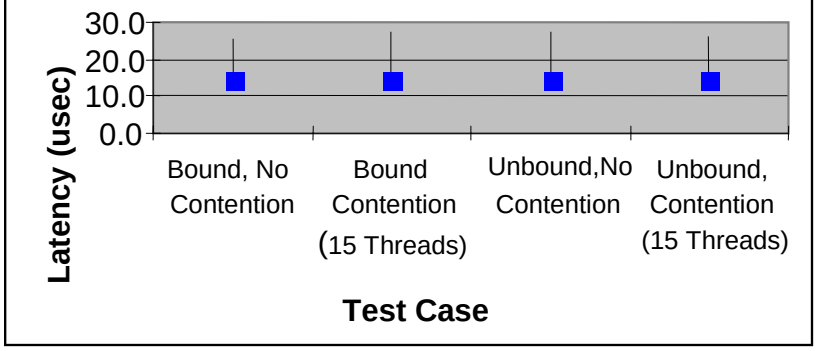


Figure 13. Event Latency

Results: BoundAsyncEventHandler was used for the first case and AsyncEventHandler was used for the second. Both the BoundAsyncEventHandler and AsyncEventHandler were acceptable for our needs since all cases met the success criteria. Figure 19 compares the BoundAsyncEventHandler and AsyncEventHandler latencies. Analysis of the data indicates that a relatively few measurement spikes were observed.



	Bound, No Contention	Bound, Contention (15 Threads)	Unbound, No Contention	Unbound, Contention (15 Threads)
Avg	14.746	14.504	14.664	14.632
Max	25.434	27.137	27.420	26.032
Min	14.434	14.200	14.415	14.406
Jitter	11.000	12.937	13.005	11.626

Figure 14. Event Latency Results (microseconds)

3.4 Memory Management

The RTSJ defines a range of different memory types to address real-time aspects of memory management and garbage collection. This section details tests with allocation throughput, entry, and exit performance for the heap, immortal, linear time (LT) memory, and variable time (VT) memory areas. The Jet Propulsion Laboratory created the allocation time and throughput time tests. In Figure 20, the memory area classes that are colored represent the classes with test results included herein.

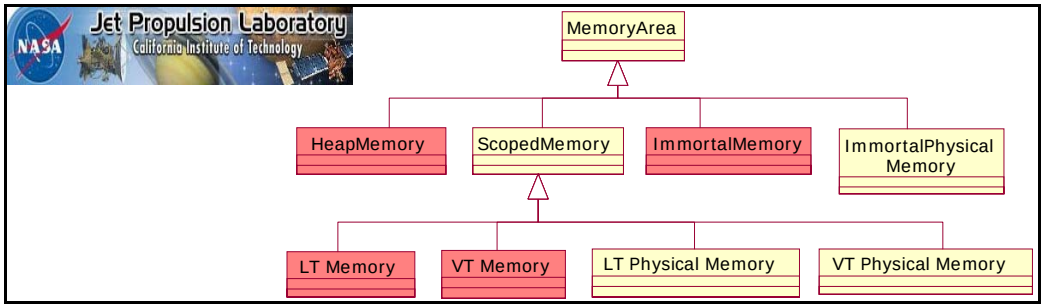


Figure 15. Memory Area Class Inheritance

3.4.1 Allocation Time vs. Memory Area

Description: Measure the time required to allocate the same sized objects in different memory areas. Place time stamps before and after the memory allocation code. Then calculate the difference between before and after times for memory allocation. Perform

this test for various object sizes from 4 to 16,384 bytes. Each thread executes at a 20-Hz periodic frame rate.

Success Criteria: Average allocation time less than 2 microseconds/byte shall be acceptable. This test is illustrated in Figure 21.

- **Description**
  - ***Measure the time required to allocated the same sized objects when running different memory area types.***
  - ***Perform this test with various sized objects.***
  - ***Average the samples per object size and memory area types and analyze results.***

Test Conditions	
Thread Type	Real-time
Memory Areas	Heap, Immortal, LT, VT
Object Sizes (bytes)	4, 8, 64, 512, 4096, and 16,384
Samples	10

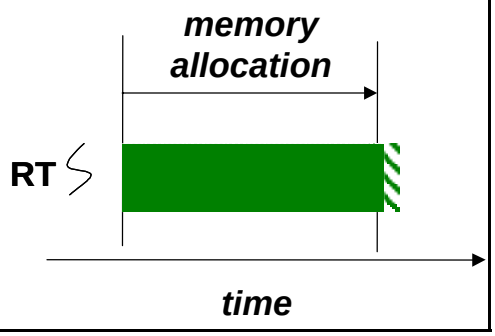


Figure 16. Allocation Time vs. Memory Area

Results: The average time to create 64 byte objects took less than 16 microseconds for all memory areas, meeting the success criteria. The times for immortal, linear time, and variable time memory areas were nearly identical for this test. See Figure 22 for per-byte allocation times in the different memory areas.

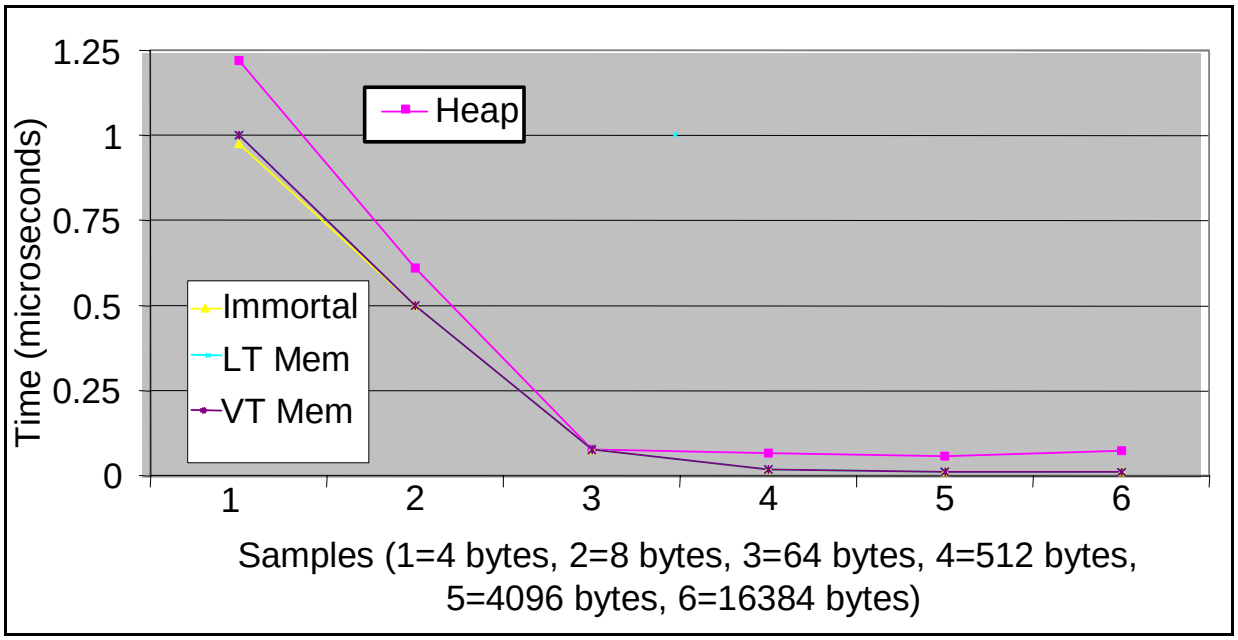


Figure 17. Maximum Memory Allocation Time per Byte for Multiple Byte Objects

3.4.2 Execution Time vs. Memory Area

Description: Measure the time needed to execute a division, logarithmic, and no operation in each memory area. The division operation is a 'divide by 2' while the log operation takes the 'log of 5'. Place time stamps before and after the call to each operation. Each thread will execute at a 20-Hz frame rate. The test is illustrated in Figure 23.

Description

- *Measure the time needed to execute various mathematical operations.*
- *Compute the maximum and minimum per operation for all memory areas.*

Test Conditions	
Thread Type	Real-time
Memory Areas	Heap, Immortal, LT, and VT
Algorithmic Functions	float division (divide by 2), logarithmic (log 5), and no operation.
Samples	10

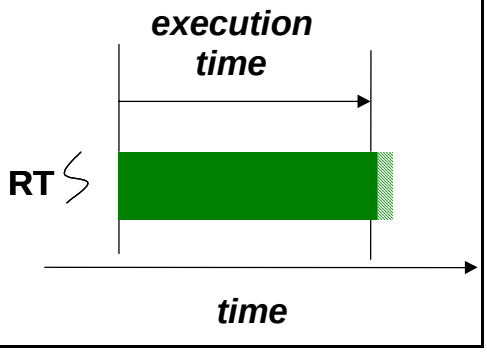
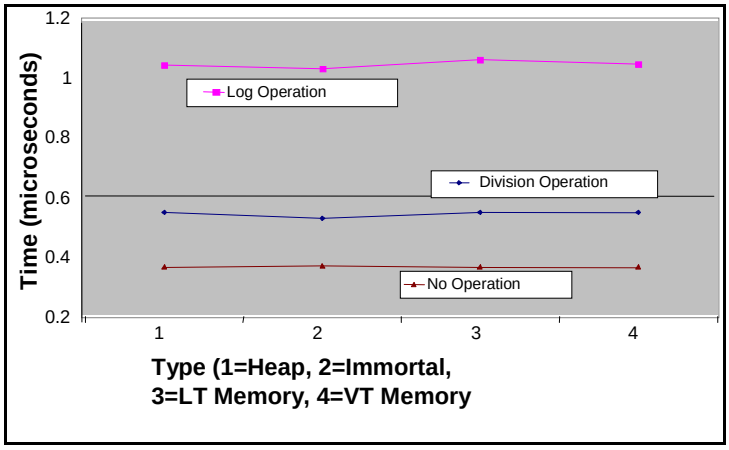


Figure 18. Execution Time vs. Memory Area

Success Criteria: The throughput values shall be within 5 per cent across all memory types.

Result: As tabulated in Figure 24, the percent variation of operation execution across all memory areas was less than 4 per cent, thus meeting our success criteria.



	Min (µs)	Max (µs)	% Delta
Division	0.5300	0.5500	4%
Log	1.0400	1.0667	2.5%
No op	0.3667	0.3700	0.9%

Figure 19. Execution Time vs. Memory Area Results

3.4.3 Memory Area Entry/Exit

Description: Log timestamps before entering a Memory Area and immediately upon exiting. Also record timestamps prior to leaving the scope and immediately after leaving the scope. Each thread will execute at a 20-Hz frame rate.

Success Criteria: Average memory area entry time shall be 100 microseconds or less. Average memory area exit time shall be 100 microseconds or less. The test is illustrated in Figure 25.

- **Description**
  - ***Measure the time required to enter into a memory a region and exit out of a memory region.***
  - ***Compute the average entry and exit times for all memory areas.***

Test Conditions	
Thread Type	Real-time
Memory Areas	Heap, Immortal, LT, VT
Samples	100

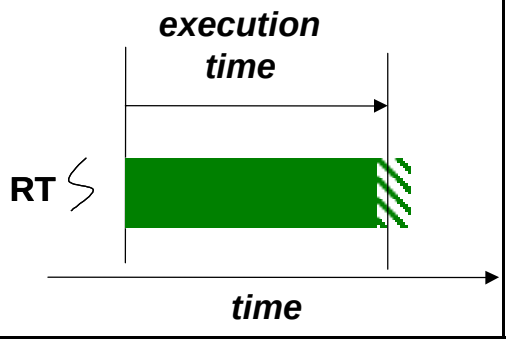


Figure 20. Memory Area Entry/Exit

Results: The average memory entry and exit times were under 20 microseconds for all measured memory types. Therefore, this test passed the success criteria. The exit times for LT memory and VT memory was substantially more than for other memory areas because the garbage collector is executed on these memory areas when their scope is freed. See Figure 26 for a graph mapping the time to enter and exit the various memory areas.

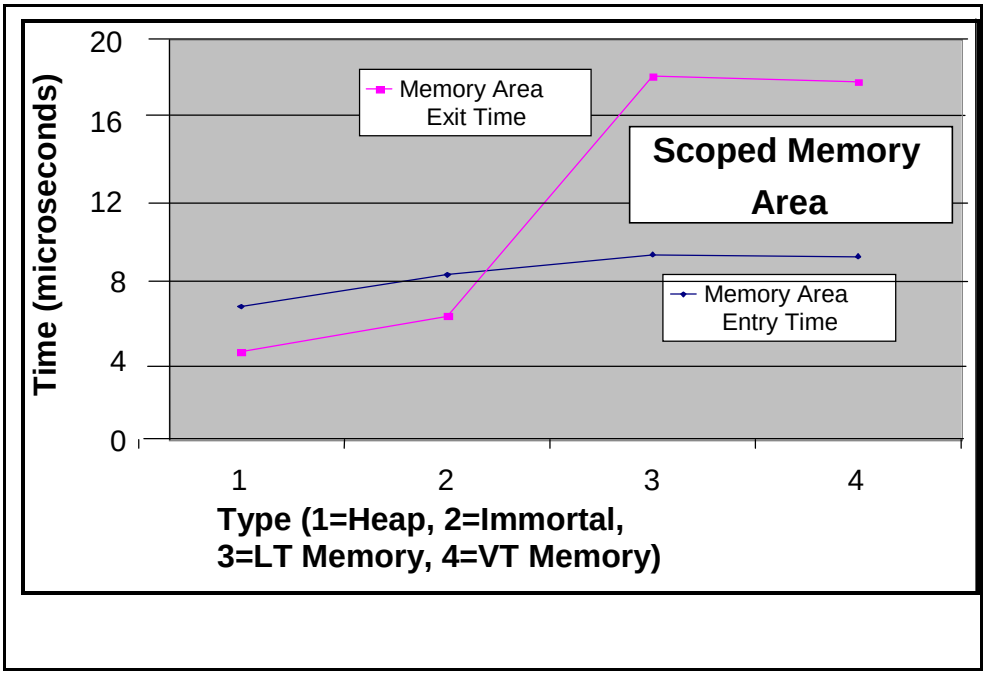


Figure 21. Memory Area Entry/Exit Results

4 Application Testing

The application test suite consists of a set of end-to-end application test scenarios and a flight control function. The end-to-end test scenarios were designed to investigate performance of component, middleware, and JVM interactions and behaviors in the context of a representative cyclically executing avionics application architecture. The flight control function was designed to investigate performance of a representative numerical algorithm.

4.1 End-to-End Application Testing

When testing the feasibility of a large-scale mission critical embedded system, a balance between a small-scale prototype and full-scale development was considered. Small-scale prototypes provide an early indication of the predicted behavior of a full-scale system. Unfortunately, costly problems sometimes occur when these prototypes are extrapolated to a large-scale system. Problems range from unexpected increase of processor throughput, increase of memory utilization, increase use of scheduling resources. A full-scale development model requires a significant amount of manpower to develop.

To balance these forces, various size scenarios were developed by combining a number of slightly modified small-scale scenarios into larger-scale scenarios. This collection of scenarios provided sufficient test coverage for predicting the behavior of a full-scale mission-critical embedded system at reduced development costs.

Leveraging technology from the DARPA Model-Based Integration of Embedded Software (MoBIES) program allowed for rapid development of these scenarios[7]. The MoBIES program includes an Open Experiment Platform (OEP) with an XML configuration framework and development tool set. The OEP provides a number of different run time product scenarios which illustrate various configurations of component-based real-time embedded systems. These scenarios contain representative component configurations and interactions, but without representative functionality to ease development and avoid classification concerns.

The MoBIES process development for Java is illustrated in Figure 27. A MoBIES configuration file in XML format was developed from the MoBIES product scenarios. A C++ translator converted the configuration file from XML to C++ header code and the Java translator converted the configuration file from XML to Java code. A Java event channel was developed utilizing the real time features of Java to replace services provided by the C++ Bold Stroke Architecture. The software components were translated from C++ to Java. A make file was used to build the C++ and Jakarta-ANT was used to build the Java. This parallel development effort provided for a side-by-side comparison of interpreted Real-Time Java to compiled C++.

[7] http://dtsn.darpa.mil/ixo/mobies.asp

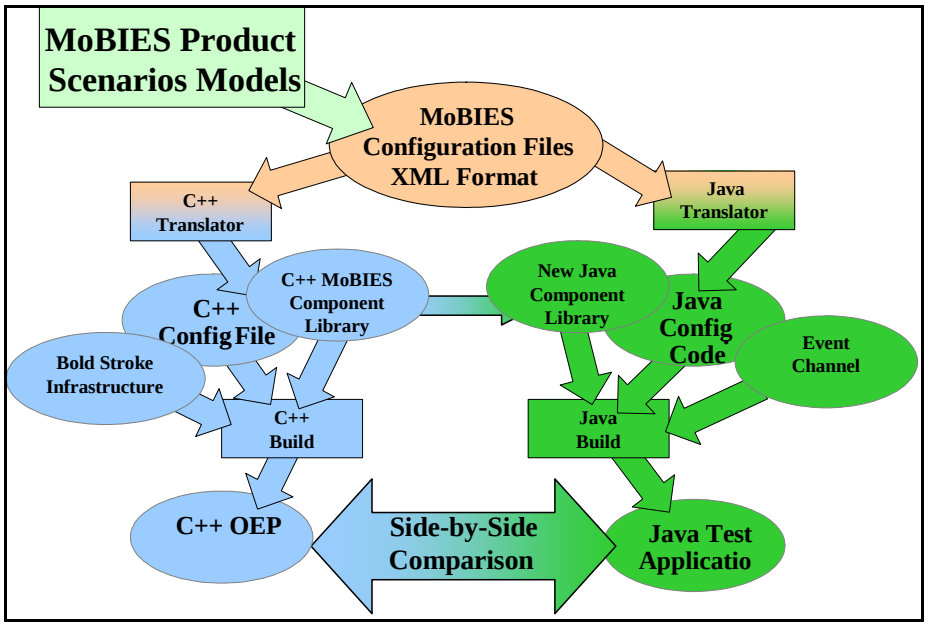


Figure 22. Leveraging of MoBIES Technology

For benchmarking purposes, a modified version of a MoBIES Product Scenario with oscillating modal behavior was selected. This product scenario has been identified as the "1X" scenario and is illustrated in Figure 28. The original version provided use of three rate group priority threads (20-Hz, 5-Hz, and 1-Hz), event correlation, and modal behavior. The benchmark version preserved these attributes and provided for a more realistic ratio of "full channel" and Event Registration Manager (ERM) event notifications[8]. Full channel events include scheduling and dispatching support, while ERM notification bypasses scheduling and provides a direct connection between supplier and consumer components for significantly increased performance. This product scenario has been identified as the "1X" scenario.

The bottom *Infrastructure* layer contains the `frameController` and the event channel. For simplicity, the event channel is not shown in Figure 28, but its role will be described here. The `frameController` is activated via a 20-Hz periodic timer and runs at the highest priority. The `frameController` pushes events to components in the *Physical Device* layer as shown via `Push()` invocations (1 through 4). These `Push()` invocations actually represent a multiple step process that includes an event supplier (e.g., `frameController`) invoking an event channel method to publish an event, the event channel determining which event consumers, if any, have subscribed to the event, and then the event channel invoking `Push()` methods on any subscriber components (e.g., `device1`). The `frameController` propagates these events at specified periodic intervals: 20-Hz for the `device1` component, 5-Hz for the `device3` and `device4` components, and 1-Hz. for the `device2` component. Since it runs at the highest rate, the `device1` component is scheduled at the highest priority in rate

[8] Tim Harrison and David Levine and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service", OOPSLA 1997, October 1997.

monotonic manner and is the second component to be activated. The `device1` component is activated from a full channel event dispatch originating from the `frameController`. The remainder of the 20-Hz components (Global Positioning System (GPS), `airframe`, and `tacticalSteering`) are executed in turn by streamlined ERM events via `Push` methods (5 through 7). The 5-Hz components on the right side of Figure 28 execute in a similar manner at a medium priority. As depicted in the middle of Figure 28, a full channel event is delivered to the `pilotControl` component only after both `device2` and `device3` have published events, resulting in a 1-Hz low-priority invocation. This combination technique is referred to as "AND" correlation. In this particular case, the two invocations shown (16 and 17) represent the publication of events to the event channel; only a single invocation to the `pilotControl` component would be made by the event channel (via 17). Each `pilotControl` execution results in a toggling between tactical steering (i.e., Mode "TAC") and navigation steering (Mode "NAV") via invocations to the `tacticalSteering` and `navSteering` components (18 and 19). The modal components (`tacticalSteering` and `navSteering`) are designed to only publish events when active. The `navDisplay` receives any event published by either of the modal steering components (8, 12, and 15), a technique referred to as "OR" correlation. When taken together, this 1X scenario provides a realistic multirate cyclic avionics execution context, but with a very small number of application components.

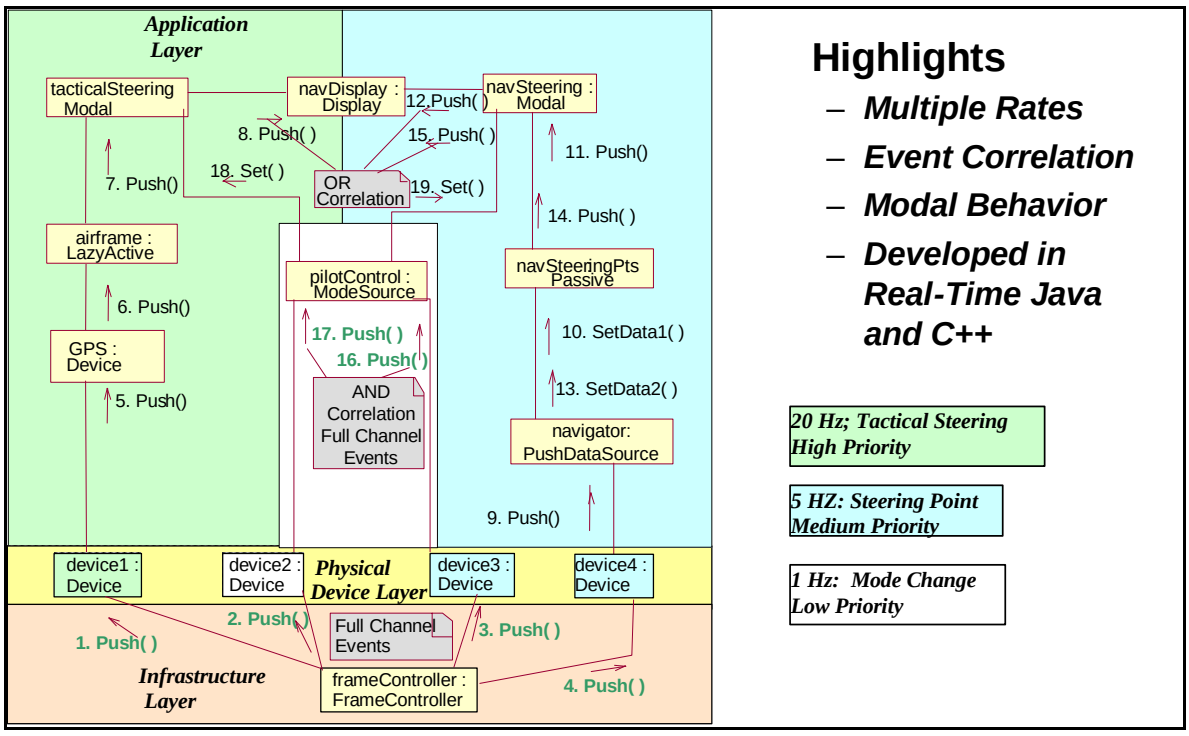
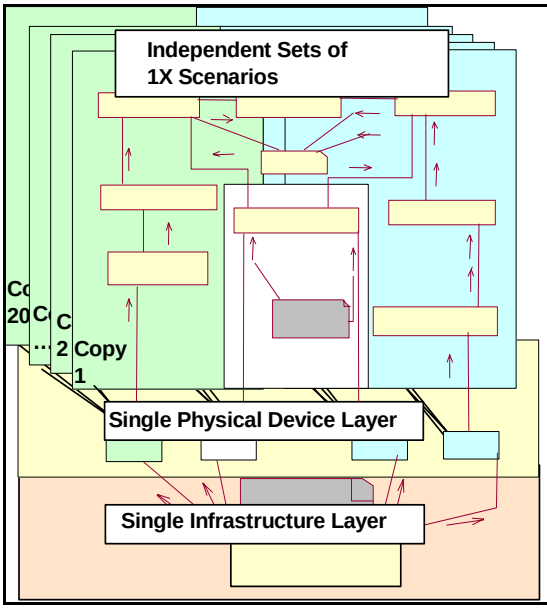


Figure 23. 1X Test Scenario

The other derived scenarios are illustrated in Figure 29. The highlighted 100X scenario contains a representative number of components and events as representative avionics mission computing systems and is used to evaluate success criteria. The Java 100X

performance was quick enough that the processing for the different rates always completed prior to the start of the next 20-Hz (fastest rate) frame, avoiding preemption. A Java 200X scenario was therefore created to measure the effects of preemption on performance and further investigate scalability. These expanded scenarios considered typical production development by increasing the number of component types and decreasing the percentage of full channel notifications as the number of component instances increase. As a result of the modal behavior of the scenario, the percentage of full channel notifications varies depending on the operation mode of the scenario.

To provide comparable tests, the preexisting MoBIES C++ application components were directly translated to Java. On the C++ side, the preexisting full-scale middleware services (e.g., CORBA real-time event service) were replaced with lightweight POSIX wrappers and prototype services that directly matched the features included in the Java event service prototype. While these implementations do not leverage language-specific features and idioms, this approach does provide a fair side-by-side comparison of interpreted Real-Time Java to compiled C++ in a realistic avionics application context.



Scenario	Component Types	Component Instances
1X	7	12
20X	14	164
50X	35	404
100X	*70*	*804*
200X	140	1604

Figure 24. Larger Scale Scenarios

A number of RTSJ features associated with scheduling and event handling are used in these tests and listed in Figure 30. RTSJ class types are listed in italics.

Object Name: Class Name	Purpose
`frameController: FrameController` (extended from *`RealtimeThread`*)	Dispatches start of rate group events to activate components in the physical device layer.
`fcPeriodics: PeriodicParameters`	Set the start time, period, cost, and deadline for the frame controller.
`fcPriority : PriorityParameters`	Sets the priority of the frame controller's periodic parameters.
fcStartTime: *RelativeTime*	Sets the start time for the frame controller's periodic parameters.
fcPeriodicTime: *RelativeTime*	Sets the period, cost, and deadline for the frame controller's periodic parameters.
eventQueue[n]: EventQueue (extended from *BoundAsyncEvent Hander*)	Stores received and dispatched events from a particular rate group. n represents the number of rate group threads.
eqAsyncEvent: *AsyncEvent*	Enables the event queue dispatcher when an event is received.
eqPriority: *PriorityParameters*	Sets the priority of the event queue.

Table 6. RTSJ Feature Usage

4.1.1 Steady-State Execution Time

Test Description: Measure the frame initiation, periodic application processing, and infrastructure processing for each test scenario. For the Java implementation, execute the processing within real-time threads (not No Heap Real-time Threads) in heap memory (not immortal memory).

Performance Success Criteria: The software system shall support execution of multiple periodic rates of application components up to 20 Hz. The execution time required for infrastructure services (e.g., middleware, JVM, operating system) shall not exceed roughly 10 per cent of the total processing time for the set of services included in this benchmark.

Performance Results: The results from the Java and C++ 100X scenarios are illustrated in Figure 31. The figure displays 16 seconds of processing time (20 samples per second for 16 seconds for 320 total samples). The 2-second cyclical (40 samples at 20 Hz) processing represents the scenario's modal behavior, with the system changing mode each second. Mode TAC requires more execution time than Mode NAV due to a greater number of components that run at the highest rate. The 20-Hz line represents the sum of the processing time within each 20-Hz period for all 20-Hz rate components. The 5-Hz line shown is obtained by summing all of the associated component execution times over a 200-millisecond period and dividing the sum by four to normalize it to the 20-Hz data. Similarly, the 1-Hz and infrastructure lines indicate the aggregate time of 1-Hz

components and infrastructure processing for a 1 second period, scaled to 20-Hz periods. Thus, within each mode, adding up the times for each rate provides the average execution time used within each 20-Hz processing frame. For example, if the beginning of the RT Java time trace is observed, approximately 0.1 + 3.0 + 4.1 + 7.6 = 14.8 milliseconds out of every 20-Hz frame are consumed by application and infrastructure execution, leaving 50 – 14.8 = 35.2 milliseconds of idle time in each frame on average.

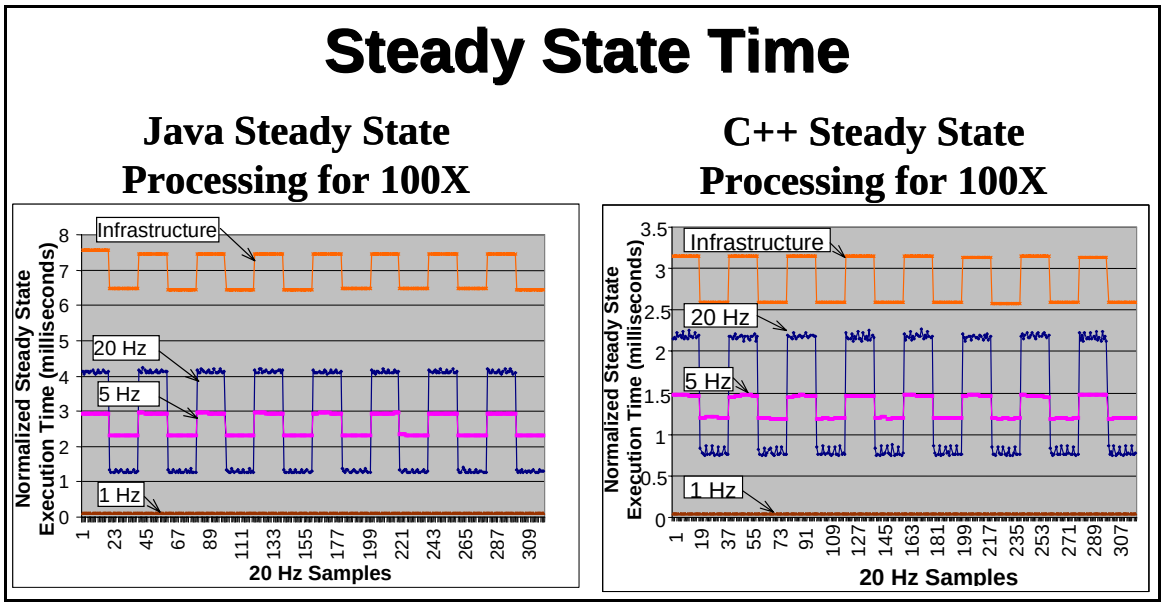


Figure 25. Steady-State Execution Time 100X Scenario

Using approximately 800 software components, the Java successfully repeated the C++ real-time behavior and properly supported the periodic rates. The 100X scenario processed approximately 600 full-channel events and 10,000 ERM events per second. All deadlines were achieved with 85.05 per cent of the processor utilization available for application processing.

The samples of peak usage of infrastructure services (i.e., while in Mode TAC) are shown in Figure 32. With the Java 100X scenario, the infrastructure services peaked at 14.95 per cent of the total processing time in Mode TAC and the C++ 100X scenario peaked at 6.30 per cent. Both the Java and C++ met the performance criteria, but the Java implementation required roughly 2.5 times as much execution time. With the 200X scenario, all periodic frame deadlines continued to be met, but higher rate processing did preempt longer-duration lower rate processing. These results indicated that both the behavior and the performance of priority preemption was acceptable for this scenario.

	Scenario	Ave	Min	Max	Deviation
Real-Time Java	1X	0.209	0.208	0.210	0.002
	20X	1.541	1.538	1.544	0.006
	50X	3.742	3.735	3.755	0.020
	100X	*7.461*	*7.453*	*7.473*	*0.019*
	200X	15.061	15.053	15.077	0.024
C++	1X	0.059	0.059	0.060	0.001
	20X	0.548	0.546	0.550	0.004
	50X	1.488	1.483	1.499	0.016
	100X	*3.144*	*3.140*	*3.152*	*0.012*
	200X	6.500	6.491	6.511	0.020

Table 7. Infrastructure During Peak Operation (milliseconds)

Determinism Success Criteria: The variability in the initiation of periodic processing frames shall not exceed 1 per cent of the associated period. The variability in the time required to process the application within each execution rate shall not exceed 5 per cent of the associated period.

Determinism Results: The variability in start of processing was less than 0.3 per cent for all scenarios of both Java and C++, easily beating the success criteria. The specific times measured between successive processing frames, and the associated jitter measured for each rate within the 100X scenario is shown in Figure 33. In addition to being within the success criteria, results indicate the deviation was evenly distributed around the average. Typically, a deviation in one processing frame was accompanied by an equally opposing deviation in the following frame (e.g., a 20-Hz frame time of 50.04 milliseconds in one frame would be immediately followed by a frame time of 49.96 milliseconds).

	Frame	Ave	Min	Max	Deviation
Real-Time Java	20 Hz	50.00	49.94	50.07	0.13
	5 Hz	200.00	199.84	200.15	0.31
	1 Hz	1000.00	999.96	1000.04	0.08
C++	20 Hz	50.00	49.98	50.01	0.03
	5 Hz	200.00	199.97	200.02	0.05
	1 Hz	1000.00	999.98	1000.02	0.04

Table 8. Start of Frame for 100X Scenario (msec)

4.1.2 Memory Usage

Test Description: Measure process memory usage of each scenario 5 times over a 20-second time span following reboot. The measurements were taken by executing a `watch ps` from the command line which reported the memory usage every 2 seconds.

Performance Success Criteria: The software system shall support execution within a memory limit of 100 MB, which is representative of associated avionics systems.

Performance Results: The results are provided in Figure 34. Both the Java and C++ implementations met the performance criteria, although a firm conclusion would require full application and middleware functionality. The C++ utilized less memory than the Java.

	Scenario	Memory Usage (megabytes)
Real-Time Java	1X	61.95
	20X	62.05
	50X	62.20
	100X	*62.48*
	200X	63.08
C++	1X	51.84
	20X	52.14
	50X	52.59
	100X	*53.36*
	200X	54.88

Table 9. Memory Usage

Determinism Success Criteria: The variability in the memory used by the software system during multiple identical runs shall not exceed 1 per cent.

Determinism Results: Within the precision of the measurement technique, there was no variation between the memory observed on five successive identical runs.

4.1.3 Duration of Operation

Test Description: Sample times for frame initiation, periodic application processing, and infrastructure processing, and memory for the 100X scenario for a ten hour period, and ensure that they remain stable.

Performance Success Criteria: The software system shall support prolonged period of operation of up to approximately ten hours.

Performance Results: The Java system successfully repeated the real-time behavior of the C++ and properly supported each of the periodic rates during the extended ten hour period. The results recorded at the end of operation from the Java scenario are illustrated in Figure 35, which match the short duration test results reported in the steady state test. Java memory usage was measured at 62.48 MB and C++ memory usage was measured at 53.37 MB, which matches the short duration test results described in memory test. The Java and C++ memory remained under the 100 MB success criteria and did not vary during steady state operation for the ten hour duration. No memory leaks or other performance degradations were observed.

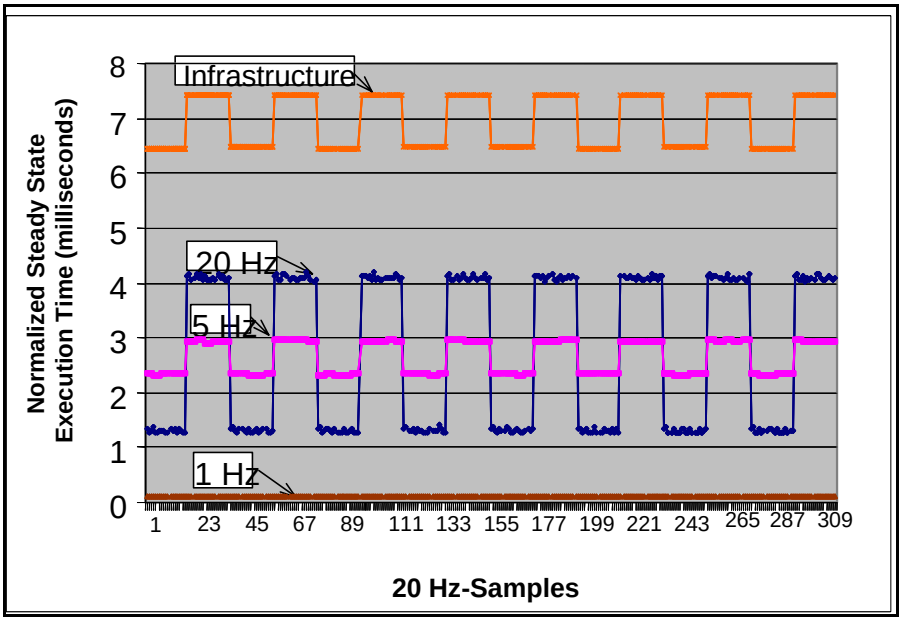


Figure 26. Duration of Operation

4.1.4 Scalability

Test Description: Measure the performance impact associated with adding software components.

Performance Success Criteria: The software system will minimize the performance cost for adding software functionality to O (n log n).

Performance Results: Collecting the results from earlier tests, the scalability results for memory and infrastructure processing during peak operation is illustrated in Figures 36. Both Java and C++ performed in roughly linear time. The slope of the infrastructure processing time for Java is roughly 2.3 times the slope of the C++ line. Conversely, the slope of the memory usage line for C++ is roughly 2.5 times the slope of the Java line.

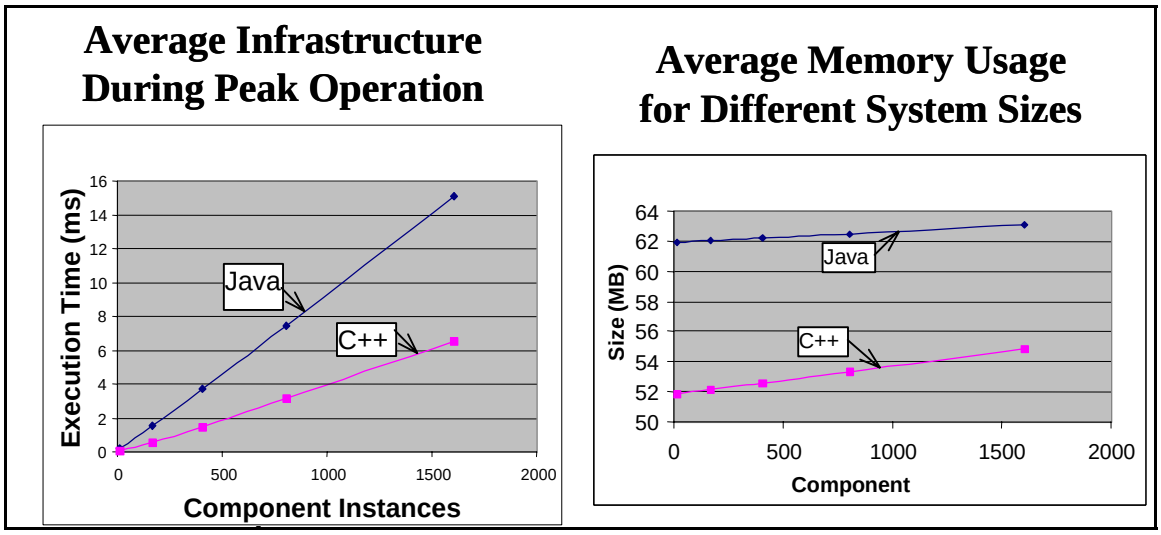


Figure 27. Scalability

4.2 Flight Control Algorithm

The flight control function contains a representative linear parameter varying (LPV) controller algorithm similar to those found in avionics embedded systems. The algorithm contains relatively heavy use of floating point operations within matrix equations. The function was translated from C++ to Java in order to provide a side-by-side comparison of the C++ and Java configurations from a computational perspective. Of special note, the Java implementation was written to avoid dynamic memory allocation during steady-state operation to match existing practice in our typical C++ implementations. This algorithm did not exploit unique real-time features, and was therefore measured in both real-time and non-real-time Java environments. The flight controls algorithm is illustrated in Figure 37.

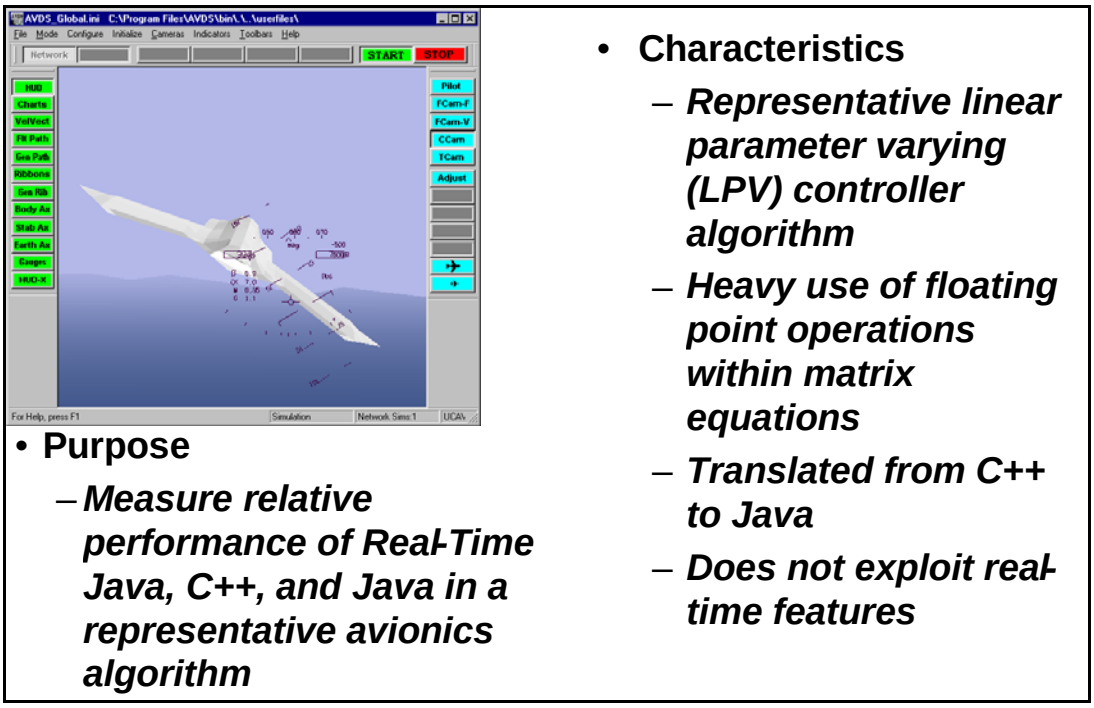


Figure 28. Flight Controls Algorithm

4.2.1 Steady-State Execution Time

Test Description: Measure the execution time for the flight controls function in a continuous loop for 1,000 samples. In addition to the C++ and RT Java configurations, include the Java J2SE with HotSpot for reference. The algorithm is run in the main program thread, with the Java implementations using heap memory

Performance Success Criteria: Since this test was created to provide a relative comparison of throughput, no specific success criteria are defined.

Performance Results: The results are shown in Figure 38. For this algorithm, the C++ implementation performed about 40 times faster than RT Java, and the Java performed about 19 times faster than RT Java on average. The wide deviation and the proximity of the average and minimum times indicates the presence of timing spikes due to the non-determinism of the plain Java implementation. Even though memory is neither allocated nor deallocated during execution, RT JVM tests were repeated in immortal memory to check for any associated effects. The RT Java average execution time using immortal memory was measured at 3.498 milliseconds. The time shown in Figure 38 is run outside of immortal memory. As expected for this test context, memory effects were minor (~0.5 per cent).

The relative execution times of the floating point operations is also shown in Figure 38. Integer operations were also investigated. The results indicated that the performance difference was not primarily due to arithmetic operations. Profiling was performed on the RT Java implementation, and no condensed portion of the flight control algorithm was found to be consuming a majority of the execution time. Given the widely scattered nature of the performance difference, the performance is speculated to be due to the interpreted nature of the RT Java implementation and overall overhead associated with RT JVM memory management.

Execution Time (milliseconds)

	Ave	Min	Max	Deviation
Real-Time Java	3.479	3.474	3.565	0.092
C++	0.085	0.085	0.097	0.013
Java	0.181	0.179	0.503	0.324

Total Floating Point Operation Execution Time Percentage.

Real-Time Java	C++	Java
3.22%	36.81%	19.7%

Figure 29. Throughput Performance

5 Real-Time Java Laboratory Demonstration

In order to show the benefits of Real-Time Java, the RTJES project teamed up with the Insertion of Embedded Infosphere Support Technologies (IEIST) project to perform a laboratory demonstration. The demonstration highlighted Real-Time Java code mobility, distribution, and portability.

5.1 Real-Time Java Demonstration Overview

A top-level overview of the Real-Time Java Demonstration is illustrated in Figure 39. The main components of this scenario are the Joint Battle Infosphere (JBI), F-15 Fighter Platform, Command and Control (C2) Platform, Force Template, Guardian Agent, and Guardian Agent Factory. The JBI represents a military network that is activated during a battle designed for military assets to publish and subscribe information in real-time. The F-15 represents one of these military assets engaged in a battlefield scenario to destroy enemy targets. The F-15 stores critical information about itself in a Force Template. The Force Template contains mission planning data and mobile code to be downloaded to a Guardian Agent Factory located on a C2 Platform. The Guardian Agent Factory is a static "always available" software entity design to create a Guardian Agent on demand. The Guardian Agent is a dynamic software entity designed to interact with the JBI to publish and subscribe information for a platform entering the battlefield scenario. This particular Guardian Agent used in this lab demonstration is for the F-15. The Guardian Agent filters the information for the F-15 so that the platform only receives the critical information required for the F-15 pilot to successfully complete his/her mission. For the lab demonstration, the Guardian Agent Route Threat Evaluator was performed in Real-Time Java. The real-time platform consisted of the following components:

1) TimeSys Linux was used as the real-time operating system.
2) TimeSys JTime was used as the real-time JVM.
3) A built in scheduler was developed to support both periodic and aperiodic scheduling.
4) The UCI Zen ORB was incorporated to provide distribution support using CORBA.
5) An event service was developed to provide decoupled cross thread communication between software components of the Route Threat Evaluator.

The basic scenario consisted of first transferring the Force Template containing the mobile code and mission planning data from the windows Guardian Agent Factory to the Real-Time Java Guardian Agent Factory. The Real-Time Guardian Agent Factory in turn instantiated the mobile code and provided a distributed component for the rest of the guardian agent to communicate.

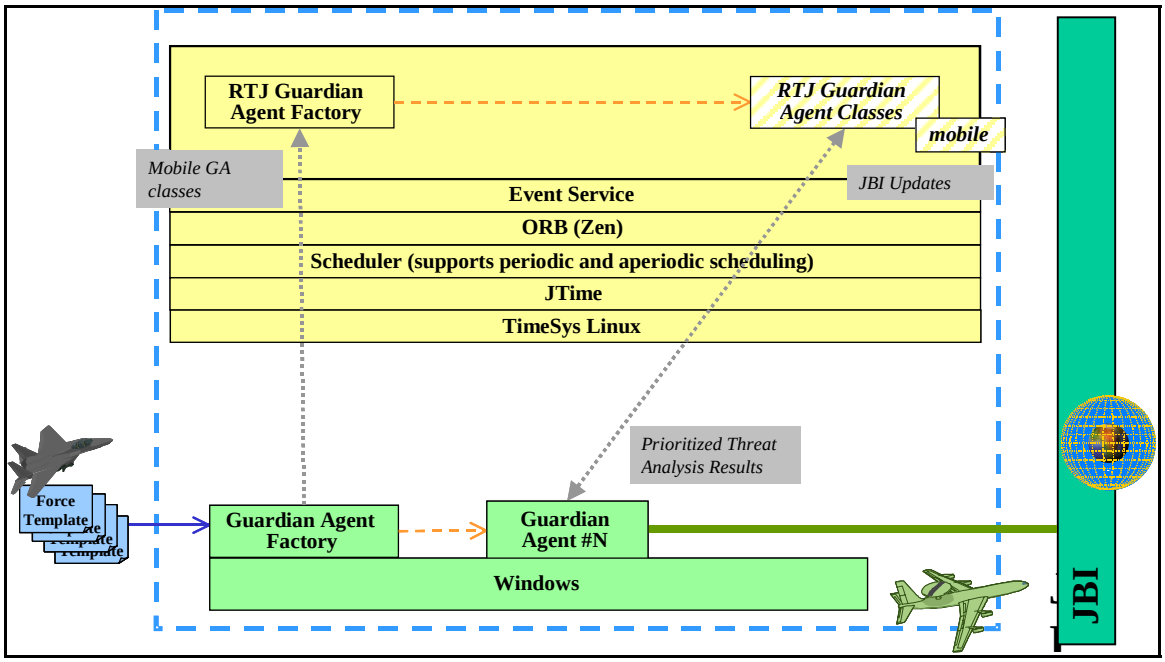


Figure 30. Real-Time Java Demonstration

5.2 Route Threat Evaluator

Figure 40, shows more of a detailed version of the internal workings of the Route Threat Evaluator (RTE). The RTE Master Server Factory generates an RTE Master used for registration of the Guardian Agent. Upon completion of Guardian Agent registration, the Guardian Agent receives a personal RTE platform server to perform communication. The RTE platform server is part of the mobile code that was transmitted during initialization.

The RTE also handles route updates. The new route is sent to the RTE platform server after registration has been completed. The route updates are scheduled at a low priority to ensure threat analysis during operation has higher priority.

The final task of the RTE is threat analysis. The RTE analyzes the position and two dimensional signature of the platform to ensure proper threat analysis is performed. The RTE provides a complete list of all the segments of the route exposed during execution and the time of impact to the exposure is also computed.

The RTE can be used for future IEIST demonstration and provides software modules that are currently being incorporated into the rest of the Guardian Agent to improve performance. The current capabilities of the RTE are listed below.

1) Dynamic Class Loader – Allows for real-time customization of the RTE for specific platform requirements.
2) Multiple Priority Scheduling – Used to provide precedence for target evaluation over other tasks like platform registration.

3) Multiple Platform Registration – Allow for multiple platforms to use the RTE for threat evaluation.

4) Two Dimensional Signature Analysis – Takes into consideration the two-dimensional signature of the platform instead of a single point solution.

5) Multiple Threat Evaluation – Tested with up to 1000 threats for a given analysis.

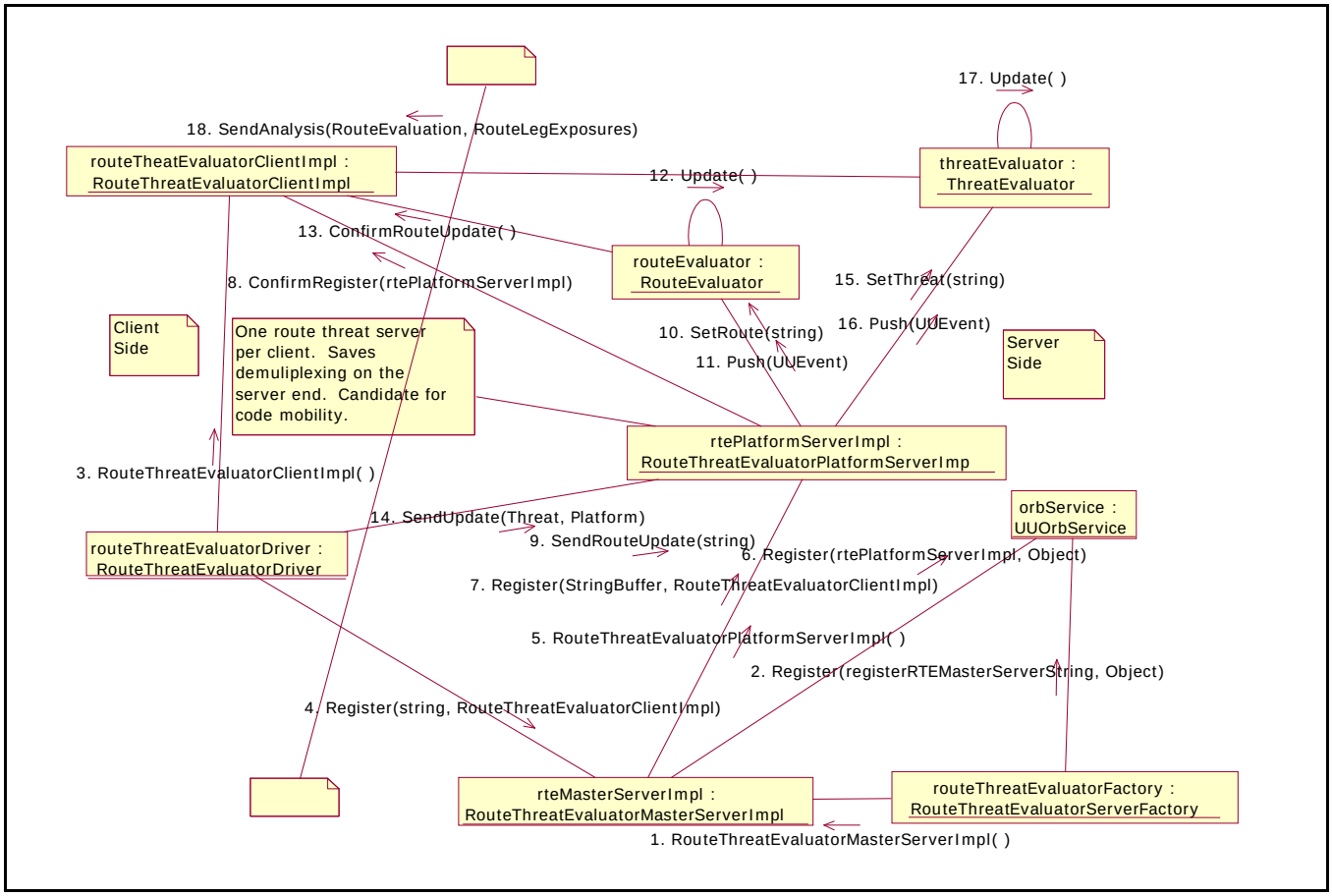


Figure 31. Route Threat Evaluator Scenario

5.3 Distribution Benchmarks

As a result of the demonstration, analysis was performed to determine the effects of distribution on the RT Java. A driver was developed to exercise the RTE under maximum conditions. The platforms registered with the RTE were varied with three test cases (1, 5, and 20 platforms) and the threats were varied in 100 threat increments from 0 threats to 1,000 threats. A side-by-side comparison was performed on regular Java using the standard Sun ORB and RT Java using the UCI Zen RT ORB. The RTE was hosted on TimeSys Linux RT OS and the driver was hosted on a non real-time Windows OS platform.

The performance results of the demonstration are illustrated in Figure 41. As shown on the graph, the standard Java was about 15 times faster than RT Java. These performance results were consistent but slightly better than early benchmarks with the Flight Controls Algorithm. The slight improvement in performance can be attributed to the efficiency improvements incorporated into the UCI Zen RT ORB. Neglecting the effects of initialization, adding more threats provided linear performance growth with the time required to produce analysis.

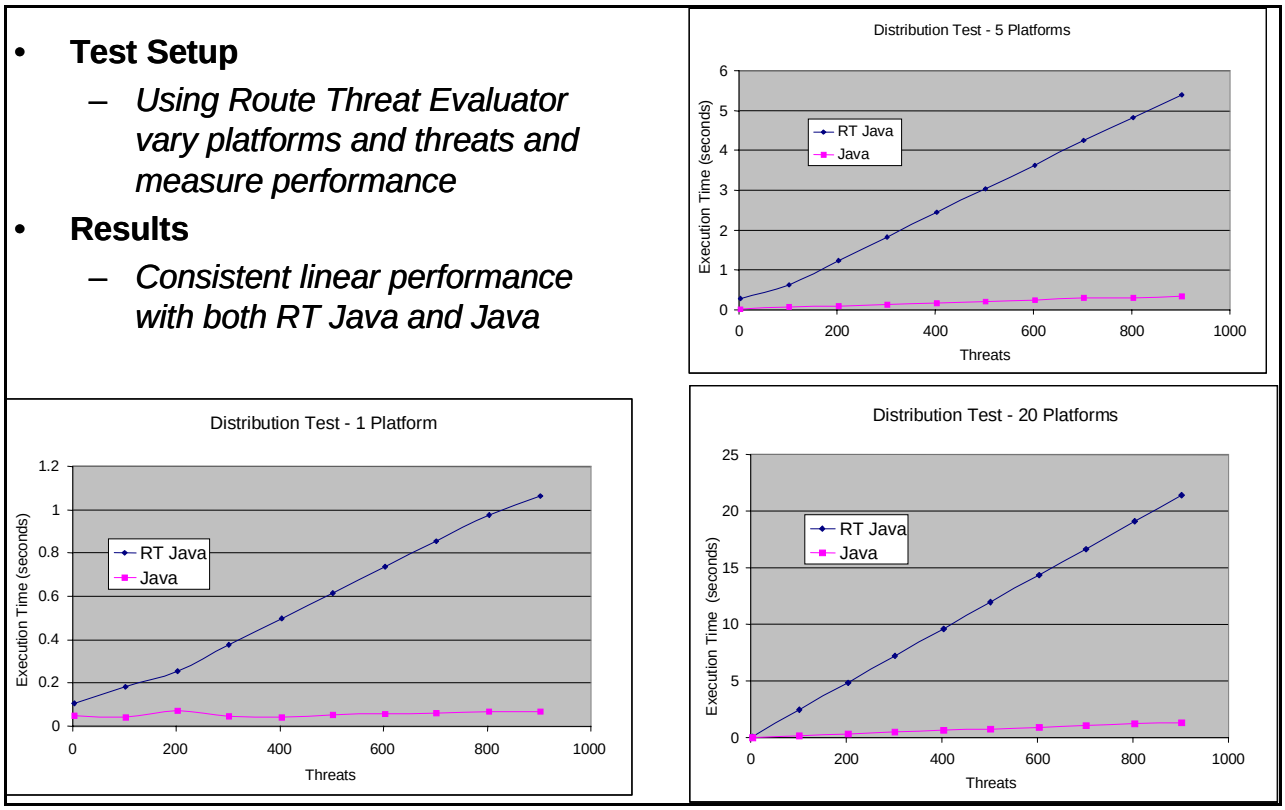


Figure 32. Distribution Benchmarks

As a final metric, the deviation was measured using 900 threats, 25 samples, and varied platform quantities. The Real-Time Java deterministic performance was significantly better than the standard Java in all cases. Even when taking into consideration the improved execution performance of the standard Java, the absolute deviation for the RT Java was better than the standard Java. These deterministic performance test results are illustrated in figure 42.

Deviation was measure using 25 samples, 900 threats, and varied platform quantities

Platforms	RT Java %	RT Java Abs Dev (ms)	Java %	Java Abs Dev (ms)
1	0.49%	5.23578	10.61%	7.54114
5	0.29%	15.78437	5.32%	18.14233
20	0.08%	16.92459	1.29%	17.18519

Table 10. Deterministic Performance of Route Threat Evaluator with 900 Threats

6 Summary

Java offers better portability, mobility, productivity, and aspect programming than other more traditional embedded software programming languages such as C++ and Ada 95. Also, Java provides interoperability with other programming languages. One requirement in the past that has precluded developers from using Java for the embedded environment has been real-time deterministic performance. The Real-Time Specification for Java (RTSJ) was written to encourage Java Virtual Machine (JVM) development to help bridge this real-time gap for the embedded software community. Unfortunately, 3 years have passed since initial release of the RTSJ, and there is only one commercially available product called TimeSys JTime Java that is RTSJ compliant. We have completed significant benchmarking on JTime and find the product to have satisfactory real-time performance but poor throughput performance. We observed significant reduction in throughput performance when using JTime compared to C++ and the standard Sun JDK Java. JTime offers Ahead Of Time (AOT) compilation to help improve performance without precluding the use of code mobility for non-precompiled parts of the code. Our experimentation with JTime AOT has demonstrated some improvement but not to the order of magnitude required for a production ready program. Also, the limited platforms available with JTime (only available on TimeSys Real-Time Linux) prevent the use of this product on most anticipated hardware platforms.

Principal remaining areas of concern for our applications include startup time, memory management, and mixed language support. Some of these investigations will require running on an embedded target platform (e.g., without file systems), which were not currently supported in our experimentation system.

Our results indicate that the basic prerequisite real-time characteristics for mission-critical avionics systems are slowly emerging in commercial implementations. We experienced great portability, mobility, productivity, and aspect programming benefits in working with Java. However, the future of RT Java in whatever application depends largely on its being embraced by the commercial market place. Without the support of these vendors, it will be at most a fringe player with performance substantially less than that of C++.

7 Reference Documents

7.1 Boeing Documents

E. Pla, J. Urnes, K. Luecke, and D. Sharp, "Test Descriptions and Results for the Real-Time Java for Embedded Systems Program," Contract Number F33615-97-D-1155, Delivery Order (DO) 0008, 17 April 2003.

7.2 Other Documents

R Klefstad, A. S. Krishna, and D.C. Schmidt, "Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, Embedded CORBA Applications, *Proceedings of the Distributed Objects and Applications (DOA) conference, Irvine, CA*, October/November 2002.

A. Corsaro and D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems," *Proceedings of the 8th IEEE Real-time Technology and Applications Symposium*, September 2002.

F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming," *OOPSLA* 2001 *Advanced Separation Of Concerns Workshop*, October 2001.

C. P. Satterthwaite, D.E. Corman, and T.S. Herm, "Transforming Legacy Systems To Obtain Information Superiority," *6th International Command and Control Research and Technology Symposium*, June 2001.

G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi, "The Real-Time Specification for Java" Addison-Wesley, 2000.

B.S. Doerr and D.C. Sharp "Freeing Product Line Architectures from Execution Dependencies," *Software Technology Conference*, May 1999.

D.C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," *Software Technology Conference*, May 1998.

D.C. Sharp, "Avionics Product Line Software Architecture Flow Policies," *AIAA/IEEE Digital Avionics Systems Conference*, October 1999.

Harrison T., D. Levine, and D.C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," *OOPSLA* 1997, October 1997

D.C. Winter, "Modular, Reusable Flight Software For Production Aircraft," *AIAA/IEEE Digital Avionics Systems Conference Proceedings*, October, 1996, pp. 401-406.

Appendix A. IEEE RTAS '03 Conference Paper

Evaluating Real-Time Java for Mission-Critical Large-Scale Embedded Systems

David C. Sharp, Edward Pla, & Kenn R. Luecke
The Boeing Company, Saint Louis, Missouri, USA
{david.sharp,edward.pla,kenn.r.luecke}@boeing.com

Ricardo J. Hassan II
Jet Propulsion Laboratory,Pasadena, California, USA
ricardo.hassan@jpl.nasa.gov

Abstract

Many of the benefits of Java, including its portability, networking support, and simplicity, are of increasing importance to large-scale distributed real-time embedded (DRE) systems, but have been unavailable due to the lack of acceptable real-time performance. Recent work establishing the Real-Time Specification for Java (RTSJ) [1] has led to the emergence of Real-Time Java Virtual Machines (RT JVMs) that promise to bridge this gap. This paper describes benchmarking results on an RT JVM. This paper extends previously published results [2] by including additional tests, by being run on a recently available pre-release version of the first commercially supported RTSJ implementation, and by assessing results based on our experience with avionics systems in other languages.

1. Introduction

The Boeing Company is currently experimenting with Real-time (RT) Java as part of the Air Force Research Laboratory (AFRL) RT Java for Embedded System (RTJES) program [1]. This program investigates the use of Java in hard and soft large-scale distributed real-time embedded (DRE) avionic system applications. The program has two primary objectives: benchmarking RT Java implementations to assess their suitability for this domain, and demonstrating the operational benefit of RT Java features for network-centric applications. This paper describes results of a portion of the network-centric benchmarking effort on a pre-release version of the commercial JTime® RT JVM from TimeSys that implements the RTSJ.

[1] This work was sponsored by the Air Force Research Laboratory, Wright-Patterson Air Force Base, Information Directorate, under contract F33615-97-D-1155-0008.

The RTSJ defines a set of classes which provide capabilities supporting real-time operation in a Java environment, including threading, scheduling, event handling, synchronization, and memory management. Specially constructed RT JVMs support the real-time semantics defined in the class library specification.

Our benchmarking efforts focus on assessing the performance and determinism of systems using these RT JVM features via two sets of tests. The first set of tests (which are discussed in this paper) assesses the characteristics of individual RTSJ features. The second set of tests (which are not discussed in this paper) investigates performance within an environment that is representative of an actual avionics application, based on our experience with reusable component-based avionics systems on the Boeing Bold Stroke initiative [3]. We plan to publish these latter test results when complete.

The remainder of this paper is organized as follows. Section 2 describes the experimental system configuration. Section 3 describes the low-level RTSJ benchmarking results. Concluding remarks and acknowledgements follow in Sections 4 and 5, respectively.

2. Experimentation System

This section describes the configuration of both the hardware and software test platform.

2.1. Hardware System

A Dell GX 150 computer was used for Java benchmark development and execution. This computer has a 1.2 Gigahertz Pentium 4 single processor. It has a 12 GB hard drive, 256 MB of RAM, 256 KB of cache memory, and 900 MB of swap memory.

2.2. Software System

As of this writing, the only known commercial implementation of the RTSJ is from TimeSys, for which we received a pre-release version. Prior to availability of this product, we developed tests and measured results on the openly available Reference Implementation (RI), also from TimeSys. The RI was designed to investigate and demonstrate the semantics of the RTSJ, not for production-quality run-time performance. Prior RI benchmarking confirmed this [2], but these results are not included here due to space constraints.

The test platform was configured with Red Hat Linux version 7.2, with real-time support provided by TimeSys Linux/NET for X86 UNI platform operating system extensions, version 3.1.214c, and TimeSys RT JVM version 3.5.3.

The JVM was executed with a memory allocation pool of 50 MB (-Xms50M). The immortal memory size was set to 80 MB (IMMORTAL_SIZE=80000000).

The tests were developed with the Jakarta Ant version 1.4.1 build tool with javac from the Java Development Kit (JDK) version 1.2.2. This javac version was selected due to version compatibility issues in libraries used in the avionics application test set. No just in time or ahead of time compilation was performed for these tests.

3. RTSJ Testing

These tests focus on assessing the performance of specific RTSJ capabilities. These tests are being added to the RTJPerf open source RT JVM benchmarking suite established by Angelo Corsaro at the University of California, Irvine (UCI) [4] and Washington University in St. Louis. Taken together, tests were created to assess determinism, latency, and throughput associated with threads, scheduling, memory management, synchronization, time, timers, asynchrony, exceptions, and class loader and dynamic linking. Only the tests deemed of most importance are included here due to space constraints.

For each test, a description of the test is included, along with success criteria, and experimental results and analysis. The success criteria are based on requirements and experiences with avionic mission computing systems. While these criteria are intentionally domain specific, they do capture expectations for an important category of embedded systems. In all cases, the raw measured values are provided for comparison against criteria in other domains.

3.1. Throughput

The RTSJ introduces two new types of threads as shown in Figure 1: RT threads and No Heap RT (NHRT) threads. RT threads support, at a minimum, basic real-time preemptive scheduling. No Heap RT threads add the guarantee that execution will be independent of garbage collection but with the additional restriction that heap-based memory not be used. This section outlines tests assessing the throughput of these different thread types in varying execution environments.

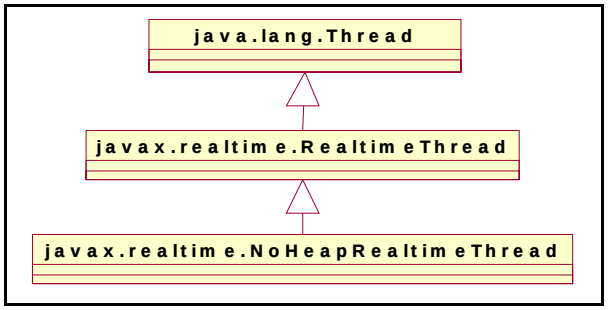


Figure 1. Thread Inheritance in RT Java

3.1.1 Thread Throughput

Description: Record the execution time of a computationally intensive algorithm, representative of avionics mission computing processing, when run in different thread types: NoHeapRealtimeThread (NHRT), RealtimeThread (RT), and java.lang.Thread. All three thread types will be processing in a 20 Hz frame. In the thread's `run()` method, log timestamps before and after the algorithm executes. This computationally intensive algorithm is a flight controls algorithm that is CPU intensive and reads data from two different input files. The flight controls algorithm performs all of its memory allocation and reference storage upon initiation. The NoHeapRealtimeThreads were executed using Immortal memory which has no garbage collection while the RealtimeThreads were executed from heap memory.

Success Criteria: Throughput in different threads shall not vary by more than 1%

Results: The throughput for NHRT, RT, and normal java threads on the selected algorithm was comparable. The largest percentage time difference between the three thread types was 0.643% for the NHRT and RT threads. See Table 1 for more throughput comparisons between NHRT, RT, and normal threads.

Table 1. Algorithm Execution Times Within Different Thread Types (milliseconds)

	Time in NHRT Threads	Time in RT Threads	Time in Normal Threads	Time (% Difference)
Avg	3.5201	3.5087	3.5174	0.324%
Max	3.5372	3.5601	3.5439	0.643%
Min	3.5109	3.4977	3.5076	0.376%

3.1.2 Thread Throughput With Contending Background Threads

Description: Record the execution time of the same mission computing algorithm, when run with different scheduling parameters and competing threads. The algorithm is CPU intensive and executes in a NoHeapRealtimeThread with a priority of 260. Measure how the same functionality scheduled with varying numbers of lower priority threads behaves. The contending threads execute in RealtimeThreads with lower priorities. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20 Hz frame rate.

Success Criteria: Difference between the tests with no background threads and the tests with 15 background threads shall be less than 5%.

Result: The first case schedules only the thread being analyzed, while the second case schedules the thread to be analyzed along with 15 background threads. The difference between the maximum, minimum, and average data points were all below 1%. This meets our success criteria. See Table 2 for detailed metrics.

Table 2. Algorithm Execution Times with Contending Background Threads (milliseconds)

	0 Other Threads	15 Other Threads	% Difference
Avg	3.5253	3.5290	0.1039%
Max	3.5395	3.5669	0.7673%
Min	3.5218	3.5244	0.0756%
Jitter	0.0177	0.0424	

3.2. Determinism

The RTSJ provides direct support for initiating functionality that needs to be run at periodic intervals either via thread scheduling or via events driven by timers. This section outlines tests investigating timing jitter associated with initiating and completing periodic activities.

3.2.1 Periodic Start of Frame Determinism

Description: Using the `PeriodicParameters` class, establish a periodic thread. Immediately after the `waitForNextPeriod()` call, log a timestamp and calculate the time between invocations. Two tests were conducted. The first test was executed with only a single 20 Hz NoHeapRealtimeThread being analyzed, while the second test was executed with the 20 Hz NoHeapRealtimeThread thread being analyzed while another fifteen lower priority RealtimeThread threads were executing at a 20 Hz frame rate. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20 Hz frame rate.

Success Criteria: Jitter shall be within 1% of the period. With a representative avionics processing rate of 20 Hz, the maximum allowable jitter is 0.5 milliseconds.

Results: The maximum jitter in both tests easily surpassed the success criteria of 0.5 milliseconds. See Table 3 for details.

Table 3. 20 Hz Frame Execution Times Measured at Frame Start Up (milliseconds)

	0 Other Threads	15 Other Threads
Avg	50.0	50.0
Max	50.0048	50.0241
Min	49.9954	49.9794
Difference	0.0094	0.0447

3.2.2 Periodic End of Frame Determinism

Description: Using the PeriodicParameters class, set up a periodic NoHeapRealtimeThread thread. Immediately after the `waitForNextPeriod()` call, execute an algorithm of significant duration but not longer than the period. After the algorithm completes, log a timestamp and calculate the difference between successive timestamps. Repeat with and without competing lower priority RealtimeThread threads as for the previous test. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20 Hz frame rate.

Success Criteria: Completion time differences shall be under 0.5 milliseconds. This represents 1% of a 20 Hz frame.

Results: The maximum jitter for 0 and 15 competing threads was 0.0282 milliseconds and 0.1441 milliseconds, respectively, which easily met the success criteria. See Table 4 for specific measurement results.

Table 4. Frame Execution Times Measured at Frame Completion (milliseconds)

	0 Other Threads	15 Other Threads
Avg	50.0000	50.0000
Max	50.0153	50.0688
Min	49.9870	49.9247
Difference	0.0282	0.1441

3.2.3 Periodic Event Determinism

Description: Measure the jitter in PeriodicTimer driven AsyncEvents. Immediately inside the handleAsyncEvent method, log a timestamp and calculate the time between invocations. The first test was executed with only a single 20 Hz NoHeapRealtimeThread being analyzed, while the second test was executed with the 20 Hz NoHeapRealtimeThread thread being analyzed while another fifteen lower priority RealtimeThread threads were executing at a 20 Hz processing rate also.

Success Criteria: Periodic event timing differences shall be under 0.5 milliseconds, 1% of a 20 Hz (50 millisecond) frame.

Results: The first case was run with the AsyncEventHandler analyzing a single thread while the second case was executed with the AsyncEventHandler analyzing a single thread with fifteen background threads. The third case was run with the BoundAsyncEventHandler analyzing one thread while the fourth case was executed with the BoundAsyncEventHandler analyzing a single thread with fifteen background threads. In all cases, the jitter met the success criteria. See Table 5 for more completion time comparisons with and without competing threads.

3.3. Latency

This section details tests assessing delays associated with context switching, synchronization, and event delivery. The RTSJ supports event-based programming for two types of event handlers: bound and unbound. A bound event handler creates one thread that is permanently bound to the handler and remains active for all event fires. An unbound event handler creates a new thread with each event fire.

Table 5. 20 Hz Frame Execution Times Measured at the Event Fire (milliseconds)

	0 Other Threads Unbound / Bound	15 Other Threads Unbound / Bound
Avg	50.0000 / 50.0000	50.0000 / 50.0000
Max	50.0667 / 50.0087	50.0908 / 50.0954
Min	49.9337 / 49.9920	49.9386 / 49.9024
Delta	0.1330 / 0.0167	0.1522 / 0.1930

3.3.1 Context Switch Latency

Description: Initiate a high priority thread and a lower priority thread. Both threads will be executing at a 20 Hz frame rate. In the higher priority thread, log a timestamp before the `yield()` in the run method. In the lower priority thread, log a timestamp after the `yield()` in the run method. Then compute the latency between the higher priority thread's timestamp and the lower priority thread's timestamp.

Success Criteria: Context switch latency shall be less than 10 microseconds.

Results: See Figure 2 for a graph of the context switch latency data samples. The results show a median of approximately 2.1 microseconds. Some of the samples spike to 2.3-2.8 microseconds, probably indicating that some processing in addition to the context switch is being run following the `yield()` call. Even with this, however, the maximum time to switch between threads was roughly 2.8 microseconds, which is better than the 10 microsecond success criteria.

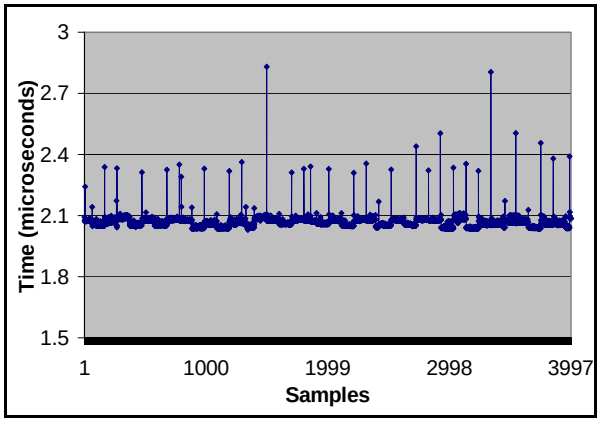


Figure 2. Context Switch Latency

3.3.2 Priority Inheritance Latency

Description: This test measures a relatively simple three thread case of priority inheritance. The low priority thread (LPT) starts and enters a synchronized method. While in that method, the medium priority thread (MPT) starts and preempts the LPT. While the MPT runs, a high priority thread (HPT) preempts the MPT and attempts to enter the same synchronized method the LPT presently has a lock on. According to priority inheritance, the LPT should get boosted up to the priority of the HPT so it can finish with the synchronized method, thus allowing the HPT to run as soon as possible. Log timestamps before and after the calls to the synchronized method. Also log timestamps at the first and last instructions inside the synchronized method. These timestamps are used to measure the boost, unboost, and total priority inheritance latency times. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Priority latency shall be under 50 microseconds for boosting and unboosting priorities combined.

Results: For both cases the maximum latency was roughly 9.0 microseconds, thus the test passed the 50 microsecond success criteria. See Table 6 for more priority inheritance boost, unboost, and total latencies.

Table 6. Priority Inheritance Latency (microseconds)

	Boost	Unboost	Latency (Boost + Unboost)
Avg	5.1372	2.7735	7.9107
Max	6.2626	3.1574	8.9635
Min	4.4778	2.6517	7.1566
Delta	1.7849	0.5057	1.8070

3.3.3 Synchronization Latency

Description: Record the time elapsed to enter a synchronized method versus a non-synchronized method. Log timestamps prior to the method call and once inside the synchronized and non-synchronized methods. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Synchronization latency shall be under 5 microseconds of overhead (difference between synchronized and non-synchronized).

Results: The test was executed for the synchronized and normal method latency cases. For each case the latency differences were less than 2 microseconds, thus this test passes the 5 microsecond threshold. See Table 7 for more synchronized and non-synchronized latencies.

Table 7. Synchronization Latency (microseconds)

	Non-Synchronized	Synchronized	Difference
Avg	1.3351	3.2385	1.9034
Max	1.7257	3.6962	1.9705
Min	1.3153	3.1975	1.8822

3.3.4 Event Latency

Description: Measure the latency from the firing of an AsyncEvent to the time it is handled. Log timestamps prior to the fire and once the event is handled. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Event latency shall be under 100 microseconds.

Results: BoundAsyncEventHandler was used for the first case and AsyncEventHandler was used for the second. Both the BoundAsyncEventHandler and AsyncEventHandler were acceptable for our needs since all cases met the success criteria. Table 8 compares the BoundAsyncEventHandler and AsyncEventHandler latencies. Analysis of the data indicates that a relatively few measurement spikes were observed as in Section 3.3.1.

Table 8. Event Latency (microseconds)

	BoundAsyncEventHandler	AsyncEventHandler
Avg	14.675	14.584
Max	27.649	27.241
Min	14.355	14.339
Delta	13.294	12.902

3.4. Memory Management

The RTSJ defines a range of different memory types to address real-time aspects of memory management and garbage collection. This section details tests with allocation throughput, entry, and exit performance for the Heap, Immortal, Linear Time (LT) Memory, and Variable Time (VT) memory areas. The Allocation Time and Throughput Time tests were created by the Jet Propulsion Laboratory. See Figure 3 for a diagram of MemoryArea inheritance relationships in the RTSJ. In Figure 3, the Memory Area classes that are colored represent the classes with test results included herein.

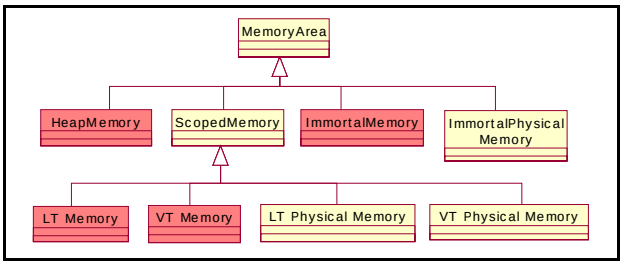


Figure 3. Memory Area Class Inheritance

3.4.1 Allocation Time vs Memory Area

Description: Measure the time required to allocate the same sized objects in different memory areas. Place time stamps before and after the memory allocation code. Then calculate the difference between before and after times for memory allocation. Perform this test for various object sizes from 4 to 16,384 bytes. Each thread executes at a 20 Hz periodic frame rate.

Success Criteria: Average allocation time less than 2 microseconds/byte shall be acceptable.

Results: The average time to create 64 byte objects took less than 16 microseconds for all memory areas, meeting the success criteria. The times for immortal, linear time, and variable time memory areas were nearly identical for this test. See Figure 4 for per-byte allocation times in the different memory areas.

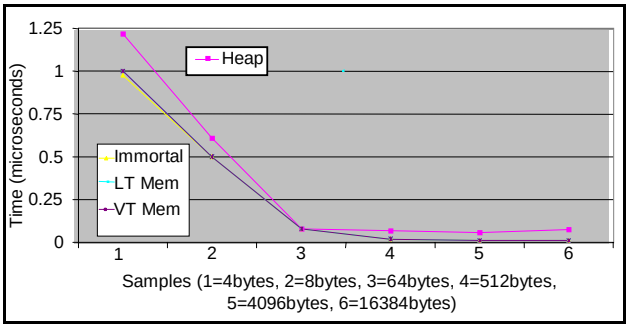


Figure 4. Maximum Memory Allocation Time per Byte for multiple byte objects

3.4.2 Throughput vs Memory Area

Description: Measure the time needed to execute a division, trigonometric, and no operation in each memory area. The division operation is a 'divide by 2' while the trigonometric operation takes the 'log of 5'. Place time stamps before and after the call to each operation. Each thread will execute at a 20 Hz frame rate.

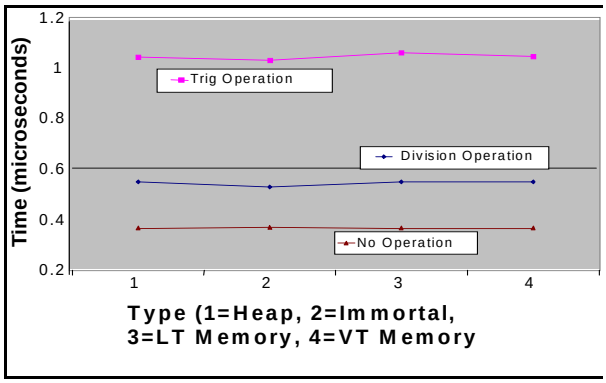


Figure 5. Operation Execution vs Memory Area

Success Criteria: The throughput values shall be within 5% across all memory types.

Table 9. Operation vs Memory Area

	Min	Max	% Delta
Float	0.5300	0.5500	4%
Trig	1.0400	1.0667	2.5%
No op	0.3667	0.3700	0.9%

Result: As tabulated in Table 9, the percent variation of operation execution across all memory areas was less than 4% thus meeting our success criteria.

3.4.3 Memory Area Entry/Exit

Description: Log timestamps before entering a MemoryArea and immediately upon entering. Also record timestamps prior to leaving the scope and immediately after leaving the scope. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Average memory area entry time shall be 100 microseconds or less. Average memory area exit time shall be 100 microseconds or less.

Results: The average memory entry and exit times were under 20 microseconds for all measured memory types. Therefore this test passed the success criteria. The exit times for LT Memory and VT Memory was substantially more than for other memory areas because the garbage collector is executed on these memory areas when their scope is freed. See Figure 6 for a graph mapping the time to enter and exit the various MemoryAreas.

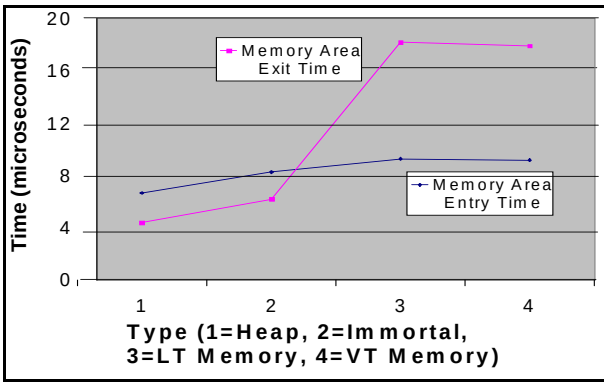


Figure 6. Average Memory Area Entry/Exit Times

4. Concluding Remarks

The experimental results in this paper indicate that emerging RTSJ implementations are capable of providing real-time characteristics with sufficient performance to meet key avionics system requirements.

There are still some areas that motivate further investigation. These areas include relative throughput to C++ or other languages, performance of memory management including garbage collection, and continued investigation into timing spikes as noted in the results. Our early results, however, indicate that the principal prerequisite real-time characteristics for mission-critical avionics systems are emerging in commercial implementations and hold promise in meeting the vision of bringing Java to large-scale DRE systems.

A second set of tests (which are not discussed in this paper) investigates performance within an environment that is representative of an actual avionics application, based on our experience with reusable component-based avionics systems on the Boeing Bold Stroke initiative. This paper will provide insight into a comparison between avionics mission computing applications that have been written in both RT Java and C++ that are based on a Bold Stroke application. We plan to publish these latter test results when complete.

5. Acknowledgements

This benchmarking effort has been a highly collaborative effort, with many contributors. We thank the US Air Force Research Laboratory Information Directorate, Wright-Patterson Air Force Base, for guiding and sponsoring this work. James M. Urnes-Jr. from Boeing created our initial set of RTSJ level tests. This paper benefited substantially from review and result interpretation insights provided by Peter Dibble at TimeSys. Ron Cytron and Ravi Pratap at Washington University in St. Louis contributed to this work, especially in the context of their aspect oriented event service named FACET, with ongoing integration results planned for future publication [5].

[1] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, R. Belliardi, "The Real-Time Specification for Java". Addison-Wesley, 2000.

[2] A. Corsaro, D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems", Proceedings of the 8th IEEE Real-time Technology and Applications Symposium, September 2002..

[3] D.C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development", *Software Technology Conference*, May 1998.

[4] A. Corsaro, D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems", Proceedings of the 8th IEEE Real-time Technology and Applications Symposium, September 2002.

[5] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming", *OOPSLA* 2001 Advanced Separation Of Concerns Workshop, Oct. 2001.

Appendix B. IEEE RTAS '03 Conference Paper

Evaluating Mission Critical Large-Scale Embedded System Performance In Real-Time Java

David C. Sharp, Edward Pla, & Kenn R. Luecke
The Boeing Company, Saint Louis, Missouri, USA
{david.sharp, edward.pla, kenn.r.luecke}@boeing.com

Abstract

Many of the benefits of Java, including its inherent portability, networking support and simplicity, are of increasing importance to large-scale distributed real-time embedded (DRE) systems, but have been unavailable due to the lack of acceptable real-time performance. Recent work establishing the Real-Time Specification for Java (RTSJ) [i] has led to the emergence of associated Real-Time Java Virtual Machines (RT JVMs) which promise to bridge this gap. This paper describes benchmarking results on a RT JVM in a uni-processing environment, and compares them to both C++ implementations of similar behavior and application requirements associated with large-scale avionics systems. This paper extends previously published results [ii][iii] by including avionics application level tests.

1. Introduction

The Boeing Company is currently experimenting with Real-Time (RT) Java as part of the Air Force Research Laboratory (AFRL) Real-Time Java for Embedded System (RTJES) program*. This program investigates the use of RT Java in hard and soft large-scale Distributed Real-Time Embedded (DRE) avionic system applications. This paper describes the results of a benchmarking effort to assess large-scale component-based avionics applications on a pre-release version of the commercial JTime RT JVM from TimeSys that implements the RTSJ.

The overall benchmarking effort is divided into two sets of tests. The first set of tests, discussed in prior work, assesses the characteristics of individual RTSJ features. The second set of tests is outlined in this paper and investigates performance within an environment that is more representative of an actual avionics application based on our experience with reusable component-based avionics systems on the Boeing Bold Stroke initiative [iv].

The remainder of this paper is organized as follows. Section 2 describes the experimental system and testing. Conclusions and Acknowledgements follow in Sections 3 and 4, respectively.

2. Experimental system and testing

This section describes the configuration of both the hardware and software elements of the test platform.

2.1. Hardware system

The same hardware, a Dell GX 150, was used for all benchmark development and execution. This computer has a single 1.2 GHz Pentium 4 processor, a 12 GB hard drive, 256 MB of RAM, and 256 KB of cache memory.

2.2. Software system

Table 1 lists the major software products used for the experiments, as well as the compiler options used that are associated with performance. All Java tests were run with bytecodes generated by Jikes being interpreted in the JVM. For associated tests, the RT JVM was executed with a memory allocation pool of 50 MB (-Xms50M). The immortal memory size was set to 10 MB (IMMORTAL_SIZE = 10000000). The options used by the C++ compiler were: `-O3 –g –fno -exceptions –fcheck –new`. The `-g` option was added to provide similar debugging capabilities to what was available via the default options were used with the Jikes compiler.

* This work was sponsored by the Air Force Research Laboratory, Wright-Patterson Air Force Base, Information Directorate, under contract F33615-97-D-1155-0008.

Table 1. Software packages

<table>
<tr><th></th><th>C++</th><th>RT Java</th><th>Java</th></tr>
<tr><td>Build Tools</td><td>GNU Make version 3.79.1</td><td colspan="2">Apache Ant Version 1.5.3</td></tr>
<tr><td>Compilers</td><td>GNU C++ version 3.1.1</td><td colspan="2">Jikes version 1.14</td></tr>
<tr><td rowspan="2">Run-Time Platform</td><td></td><td>TimeSys RT JVM version 3.5.3</td><td rowspan="2">Java J2SE with HotSpot version 1.4.1-2</td></tr>
<tr><td colspan="2">TimeSys Linux/NET for X86 UNI version 3.1.214c</td></tr>
</table>

To minimize non-deterministic system effects, all unnecessary operating services were stopped, virtual memory was disabled, and the system was rebooted in between tests. Timing was begun immediately upon the first execution of each test, without any "pre-test warm up" period.

The application test suite consists of a flight control function and a set of end-to-end application test scenarios. The flight control function was designed to investigate performance of a representative numerical algorithm. The end-to-end test program was designed to investigate performance of component, middleware, and JVM interactions and behaviors in the context of a representative cyclically executing avionics application architecture. Application component functionality was not designed to be representative of actual avionics applications for these tests; the focus was on representative application, JVM, and middleware interactions.

2.3. Flight control algorithm test and results

The flight control function contains a representative linear parameter varying (LPV) controller algorithm similar to those found in embedded avionics systems. The algorithm contains relatively heavy use of floating point operations within matrix equations. The function was translated from C++ to Java in order to provide a side-by-side comparison of the C++ and Java configurations from a computational perspective. Of special note, the Java implementation was written to avoid dynamic memory allocation during steady-state operation to match existing practice in our typical C++ implementations. This algorithm did not exploit unique real-time features, and was therefore measured in both real-time and non-real-time Java environments for comparison purposes.

Test Description: Measure the execution time for the flight controls function in a continuous loop for 1000 samples. In addition to the C++ and RT Java configurations, include the Java J2SE with HotSpot for reference. The algorithm is run in the main program thread, with the Java implementations using heap memory. Execution performance results for different RTSJ thread types is described in [3].

Performance Success Criteria: Since this test was created to provide a relative comparison of throughput, no specific success criteria are defined.

Performance Results: The results are shown in Table 2. For this algorithm, the C++ implementation performed about 40 times faster than RT Java and about 2 times faster than Java on average. The wide deviation and the proximity of the average and minimum times indicates the presence of timing spikes due to the non-determinism of the plain Java implementation. Even though memory is neither allocated nor deallocated during execution, RT JVM tests were repeated in immortal memory to check for any associated effects. The RT Java average execution time using immortal memory was measured at 3.498 milliseconds. The time shown in Table 2 is run outside of immortal memory. As expected for this test context, memory effects were minor (~0.5%).

Table 2. Flight controls function execution time (milliseconds)

	Ave	Min	Max	Deviation
RT Java	3.479	3.474	3.565	0.092
C++	0.085	0.085	0.097	0.013
Java	0.181	0.179	0.503	0.324

To better understand the source of the performance differences between the different environments, associated floating point operations were investigated. The flight controls function contains 120 multiply, 52 divide, 118 add, and 97 subtract floating point operations. Table 3 contains the execution times of floating point operations in the three different environments on the test platform.

Table 3. Individual floating point operation execution time (nanoseconds)

Operation	RT Java	C++	Java
Multiply	352.5	128.6	168.7
Divide	349.3	127.6	176.8
Add	239.4	43.1	59.4
Subtract	239.4	42.6	59.4

Based on the specified numbers of each type of floating point operation, Table 4 contains the total execution time attributable to floating point operations in each environment as both an absolute value and as a percentage of the complete execution time. These results

indicate that floating point operations consume a small fraction of the time in the RT Java case, and that floating point operation is therefore not the determining factor in the performance difference between the three environments. Floating point time is a significant portion of the execution in the other two environments, however.

Table 4. Total floating point operation execution time (microseconds / % of total execution time)

RT Java	C++	Java
111.9 (3.22%)	31.3 (36.81%)	42.2 (19.7%)

Because floating point performance did not represent the deciding factor, integer performance was investigated. Due to large number of integer operations that are performed implicitly for array indexing and other operations, determining the number of integer operations of each type was not attempted. Table 5 contains the timing associated with integer operations on the test platform, obtained by timing a large number of consecutive operations on prime numbers. In the worst case (subtraction), RT Java performance was ~5.6 times slower than C++. Since this is significantly less than the overall performance factor for the complete flight controls algorithm (i.e. 40 times slower for RT Java), integer performance is also not judged to be the source of the performance difference.

Table 5. Individual integer operation execution time (nanoseconds)

Operation	RT Java	C++	Java
Multiply	58.0	11.6	19.0
Divide	90.5	12.9	45.9
Add	42.3	2.9	14.5
Subtract	42.1	4.8	14.5

The execution time difference is therefore not primarily due to arithmetic performance. Profiling was performed on the RT Java implementation, and no condensed portion of the flight control algorithm was found to be consuming a majority of the execution time. One known overhead contributor is the RT Java bytecode interpreter. Future investigations using ahead of time compilation are planned to determine whether or not this is the primary cause or whether some latent cause is present.

2.4. End-to-end application tests and results

For the end-to-end application tests, success criteria have been derived from key operational metrics based on our experience with the required run-time performance of a range of avionics systems. A mock-up operational flight program was developed with 804 software components instances, oscillating modal behavior, three rate group priority threads (20Hz, 5Hz, and 1Hz), 604 event pushes per second, and 5% event correlation [v]. Further details exceed space limitations here. This program represents the size and event behavior of a typical avionics mission computing program. The RT Java classes used during the test were the *RealtimeThread, Periodic Papameters, RelativeTime, BoundAsyncEventHandler, and AsyncEvent.*

Test Description: Measure the frame initiation, periodic application processing, and infrastructure processing for each test scenario. For the Java implementation, execute the processing within real-time threads (not No Heap Real-time Threads) in heap memory (not immortal memory).

Performance Success Criteria: The software system shall support execution of multiple periodic rates of application components up to 20 Hz. The execution time required for infrastructure services (e.g. middleware, JVM, operating system) shall not exceed roughly 10% of the total processing time for the set of services included in this benchmark.

Performance Results: The results from the Java and C++ 100X scenarios are illustrated in Figures 3 and 4, respectively. Both figures display 16 seconds of processing time (20 samples per second for 16 seconds for 320 total samples). The two-second cyclical (40 samples at 20 Hz) processing represents the scenario's modal behavior, with the system changing mode each second. The 20 Hz line represents the sum of the processing time within each 20 Hz period for all 20 Hz rate components. The 5 Hz line shown is obtained by summing all of the associated component execution times over a 200 millisecond period and dividing the sum by four to normalize it to the 20 Hz data. Similarly, the 1 Hz and infrastructure lines indicate the aggregate time of 1 Hz components and infrastructure processing for a 1 second period, scaled to 20 Hz periods. Thus, within each mode, adding up the times for each rate provides the average execution time used within each 20 Hz processing frame. For example, if the beginning of the RT Java time trace is observed, approximately 0.1 + 3.0 + 4.1 + 7.6 = 14.8 milliseconds out of every 20 Hz frame are consumed by application and infrastructure execution, leaving 50 – 14.8 = 35.2 milliseconds of idle time in each frame on average.

The Java successfully repeated the C++ real-time behavior and properly supported the periodic rates. All deadlines were achieved with 85.05% of the processor utilization available for application processing.

Additional application testing investigated performance in a number of other areas, including steady

state determinism of periodic execution, memory usage, long-duration execution, and scalability. These tests also performed well within acceptable limits. The results of these tests have been omitted due to space constraints.

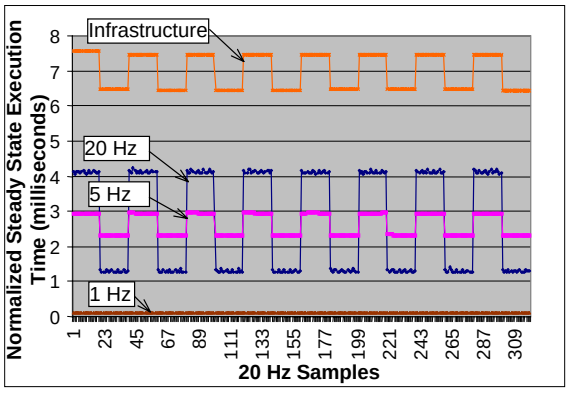


Figure 1. RT Java steady state execution time

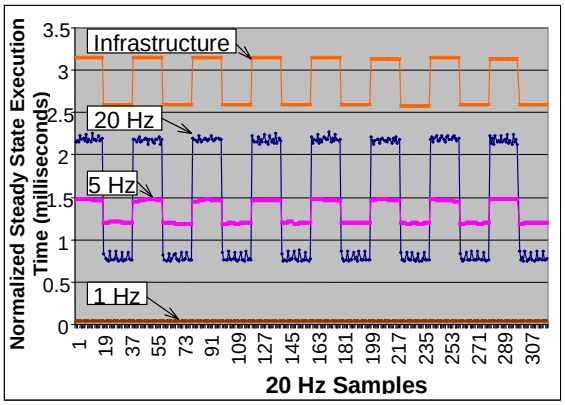


Figure 2. C++ steady state execution time

3. Conclusion

In general, our experiments indicate that emerging RTSJ implementations are capable of providing the real-time characteristics necessary to meet key avionics system requirements considered in this paper, even when applications exceed the number of components found in many avionics systems. In general, measured throughput was slower for Java, partially attributable to bytecode interpretation in this experimentation configuration. A recently available ahead of time compiler focuses on mitigating these effects, but was not included in these experiments. These results showed significant progress towards practical applicability from earlier RTSJ implementations. As typically experienced in business applications, the Java implementations were judged significantly easier to develop than the C++ implementations, although RTOS and RT JVM configuration proved somewhat challenging. Java development speed was significantly aided by faster system build times due to lack of full ahead of time compilation.

Principal remaining areas of concern for our applications include throughput, start-up time, memory management, distribution, and mixed language support. Some of these investigations would require running on an embedded target platform (e.g., without filesystems) which were not currently supported in our experimentation system. Our results, however, indicate that the basic prerequisite real-time characteristics for mission critical avionics systems are emerging in commercial implementations and hold promise in meeting the vision of bringing Java to large-scale real-time embedded systems.

4. Acknowledgements

This benchmarking effort has been a highly collaborative effort, with many contributors. We thank the US Air Force Research Laboratory Information Directorate, Wright-Patterson Air Force Base, for guiding and sponsoring this work. The application tests here were based on prior work performed on the DARPA Model-based Integration of Embedded Software program. This paper benefited substantially from review provided by Peter Dibble at TimeSys. We thank James M. Urnes-Jr., James McDonald, Dennis Noll, and Dave Lee from Boeing. Ron Cytron and Ravi Pratap at Washington University in St. Louis contributed to this work, especially in the context of their aspect oriented event service named Framework for Aspect Composition for an EvenT channel (FACET) [v].

[i] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, R. Belliardi, "The Real-Time Specification for Java". Addison-Wesley, 2000.
[ii] A. Corsaro, D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems", Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium, San Jose, CA, September 2002.
[iii] D.C. Sharp, E. Pla, K.R. Luecke, "Evaluating Real-Time Java for Mission Critical Large-Scale Embedded System", Real-Time and Embedded Technology and Applications Symposium, Washington, DC, May 2003.
[iv] D.C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development", *Software Technology Conference*, May 1998.
[v] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming", *OOPSLA* 2001 Advanced Separation Of Concerns Workshop, Oct. 2001.

List of Acronyms and Abbreviations

Acronym		Description
AFRL	-	Air Force Research Laboratory
ANT	-	Another Neat Tool
AOP	-	Aspect Oriented Programming
AOT	-	Ahead Of Time Compilation
API	-	Application Programming Interface
CORBA	-	Common Object Request Broker Architecture
COTS	-	Commercial Off The Shelf
DARPA	-	Defense Advance Research Projects Agency
ERM	-	Event Registration Manager
FACET	-	Framework for Aspect Composition for an EvenT Channel
GUI	-	Graphical User Interface
IEIST	-	Insertion of Embedded Infosphere Support Technologies
JBI	-	Joint Battle Infosphere
JCP	-	Java Community Process
JDK	-	Java Development Kit
JPL	-	Jet Propulsions Laboratory
JSR	-	Java Specification Request
JVM	-	Java Virtual Machine
LT	-	Linear Time
MoBIES	-	Model-Based Integration of Embedded Software
NAV	-	Navigation Steering
NCO	-	Network Centric Operations
NHRT	-	No Heap Real-Time
OEP	-	Open Experiment Platform
OFP	-	Operational Flight Program
ORB	-	Object Request Broker
POSIX	-	Portable Operating Systems Interface Standard
RI	-	Reference Implementation
RT	-	Real-Time
RTAS	-	Real-Time and Embedded Technology and Application Symposium
RTE	-	Route Threat Evaluator
RTJES	-	Real-Time Java for Embedded Systems
RTSJ	-	Real-Time Specification for Java
RTSS	-	International Real-Time Systems Symposium
TAC	-	Tactical Steering
TCK	-	Technology Compatibility Kit
TIM	-	Technical Interchange Meeting
TR	-	Technical Report
UCI	-	University of California, Irving
VT	-	Variable Time
WSSTS	-	Weapon Systems Software Technology Support
WU	-	Washington University
XML	-	Extensible Markup Language